

Testing

Contents

Articles

Program animation	1
Software testing	3
Portal:Software Testing	17
Accelerated stress testing	19
Acceptance testing	20
Ad hoc testing	24
Agile testing	25
All-pairs testing	26
American Software Testing Qualifications Board	27
API Sanity Autotest	28
Association for Software Testing	29
Attack patterns	30
Augmented Reality-based testing	34
Australian and New Zealand Testing Board	36
Automated Testing Framework	37
Avalanche (dynamic analysis tool)	38
User:Awright415/Centercode	39
Bebugging	40
Behavior Driven Development	41
Black-box testing	45
Block design	46
Boundary case	49
Boundary testing	49
Boundary-value analysis	49
Browser speed test	50
BS 7925-1	52
BS 7925-2	52
Bug bash	53
Build verification test	53
CA/EZTEST	54
Cause-effect graph	55
Characterization test	56
Cloud testing	57
Code coverage	58

Code integrity	63
Codonomicon	64
Compatibility testing	67
Component-Based Usability Testing	68
Conference Room Pilot	70
Conformance testing	71
Core Security	73
Corner case	76
Daikon (system)	76
Data-driven testing	77
Decision table	78
Decision-to-decision path	81
Design predicates	81
Development, testing, acceptance and production	83
DeviceAnywhere	84
Dry run (testing)	86
Dynamic program analysis	87
Dynamic testing	88
Edge case	89
Endeavour Software Project Management	90
Equivalence partitioning	92
Error guessing	93
Exploratory testing	94
Fagan inspection	96
Fault injection	99
Financial tester	104
Framework for Integrated Test	105
Functional testing	106
Functionality assurance	106
Fuzz testing	107
Game testing	111
Google Guice	116
Graphical user interface testing	117
Hybrid testing	121
IBM Product Test	122
IBM Rational Quality Manager	123
IEEE 829	126
Independent software verification and validation	128

Installation testing	129
Integration testing	130
Integration Tree	131
International Software Testing Qualifications Board	132
International Software Testing Qualifications Board Certified Tester	133
JSystem	135
Keyword-driven testing	141
Learnability	143
Lightweight software test automation	144
Load testing	145
Localization testing	148
Manual testing	149
Matrix of Pain	151
Mauve (test suite)	152
Metasploit Project	153
Microsoft Reaction Card Method (Desirability Testing)	156
Mobile Device Testing	159
Mockito	160
Model-based testing	161
Modified Condition/Decision Coverage	165
Modularity-driven testing	166
Monkey test	167
Month of bugs	168
Mutation testing	169
National Software Testing Laboratories	172
NMock	173
Non-functional testing	173
Non-Regression testing	174
Operational Acceptance Testing	177
Oracle (software testing)	178
Original Software	179
Oulu University Secure Programming Group	180
Pair Testing	181
Parameter validation	182
Partial concurrent thinking aloud	183
Penetration test	185
Performance testing	189
PlanetLab	190

Playtest	191
Portability testing	191
Probe effect	191
Program mutation	192
Protocol implementation conformance statement	195
Pseudolocalization	196
Pychecker	197
Pylint	198
User:Mbarberony/sandbox	198
Software quality	207
Recovery testing	216
Regression testing	217
Release engineering	219
Retrofits in testing	220
Reverse semantic traceability	221
Risk-based testing	225
Robustness testing	226
San Francisco depot	227
Sandbox (software development)	228
Sanity testing	229
Scalability testing	231
Scenario testing	231
Security bug	232
Security testing	233
Semantic decision table	235
Serenity Code Coverage	237
Session-based testing	238
SigmatationTF	240
Smoke testing	240
Soak testing	243
Soapsonar	244
SOASTA	245
Software performance testing	246
Software testability	252
Software testing controversies	253
Software testing life cycle	256
Software testing outsourcing	258
<i>Software Testing, Verification & Reliability</i>	260

Software verification	261
Sputnik (JavaScript conformance test)	263
STAR (Conference)	265
Stream X-Machine	266
Stress testing	268
Stress testing (software)	271
System integration testing	272
System testing	274
Tessy (Software)	275
Test Anything Protocol	276
Test automation	279
Test automation framework	283
Test automation management tools	284
Test bench	285
Test case	286
Test data	288
Test design	289
Test Double	290
Test effort	291
Test execution engine	292
Test harness	294
Test management	295
Test Management Approach	296
Test plan	300
Test script	302
Test strategy	303
Test stubs	306
Test suite	307
Test Template Framework	308
Test Vector Generator	312
Test-driven development	312
Test-Driven Development by Example	319
Testbed	319
Tester driven development	320
Tester forum	321
Testing as a service	322
Testing Maturity Model	323
Testware	324

Think aloud protocol	325
Tiger team	326
Tosca (Software)	327
TPS report	330
TPT (Software)	332
Traceability matrix	336
Tree testing	337
TTCN-3	338
Twist (software)	340
Unit testing	341
Unusual software bug	346
Usability testing	350
Utest	355
Verification and Validation (software)	358
Volume testing	360
Vulnerability (computing)	360
Web testing	368
White-box testing	370
Windmill (testing framework)	371
X-Machine Testing	371

References

Article Sources and Contributors	377
Image Sources, Licenses and Contributors	386

Article Licenses

License	387
---------	-----

Program animation

Program animation or **Stepping** refers to the very common debugging method of executing code one "line" at a time. The programmer may examine the state of the program, machine, and related data *before and after* execution of a particular line of code. This allows evaluation the effects of that statement or instruction in isolation and thereby gain insight into the behavior (or misbehavior) of the executing program. Nearly all modern IDEs and debuggers support this mode of execution. Some Testing tools allow programs to be executed step-by-step optionally at either source code level or machine code level depending upon the availability of data collected at compile time.

History

Instruction **stepping** or **single cycle** also referred to the related, more microscopic, but now obsolete method of debugging code by stopping the processor clock and manually advancing it one cycle at a time. For this to be possible, three things are required:

- A control that allows the clock to be stopped (e.g. a "Stop" button).
- A second control that allows the stopped clock to be manually advanced by one cycle (e.g. An "instruction step" switch and a "Start" button).
- Some means of recording the state of the processor after each cycle (e.g. register and memory displays).

On the IBM System 360 processor range, these facilities were provided by front panel switches, buttons and banks of neon lights.

Other systems such as the PDP-11 provided similar facilities, again on some models. The precise configuration was also model-dependent. It would not be easy to provide such facilities on LSI processors such as the Intel x86 and Pentium lines, owing to cooling considerations.

As multiprocessing became more commonplace, such techniques would have limited practicality, since many independent processes would be stopped simultaneously. This led to the development of proprietary software from several independent vendors that provided similar features but deliberately restricted breakpoints and instruction stepping to particular application programs in particular address spaces and threads. The program state (as applicable to the chosen application/thread) was saved for examination at each step and restored before resumption, giving the impression of a single user environment. This is normally sufficient for diagnosing problems at the application layer.

Instead of using a physical stop button to suspend execution - to then begin stepping through the application program, a breakpoint or "Pause" request must usually be set beforehand, usually at a particular statement/instruction in the program (chosen beforehand or alternatively, by default, at the first instruction).

To provide for full screen "animation" of a program, a suitable I/O device such as a video monitor is normally required that can display a reasonable section of the code (e.g. in dis-assembled machine code or source code format) and provide a pointer (e.g. <==) to the current instruction or line of source code. For this reason, the widespread use of these full screen animators in the mainframe world had to await the arrival of transaction processing systems - such as CICS in the early 1970's and were initially limited to debugging application programs operating within that environment. Later versions of the same products provided cross region monitoring/debugging of batch programs



System/360 (Model 65) operator's console, with register value lamps and toggle switches and buttons (middle of picture) .

and other operating systems and platforms.

With the much later introduction of Personal computers from around 1980 onwards, integrated debuggers were able to be incorporated more widely into this single user domain and provided similar animation by splitting the user screen and adding a debugging "console" to provide programmer interaction.

Borland Turbo Debugger was a stand-alone product introduced in 1989,^[1] that provided full-screen program animation for PC's. Later versions added support for combining the animation with actual source lines extracted at compilation time.

Techniques for program animation

There are at least three distinct software techniques for creating 'animation' during programs execution.

- **instrumentation** involves adding additional source code to the program at compile time to call the animator before or after each statement to halt normal execution.
- **Induced interrupt** This technique involves forcing a breakpoint at certain points in a program at execution time, usually by altering the machine code instruction at that point (this might be an inserted system call or deliberate invalid operation) and waiting for an interrupt. When the interrupt occurs, it is handled by the testing tool to report the status back to the programmer. This method allows program execution at full speed (until the interrupt occurs) but suffers from the disadvantage that most of the instructions leading up to the interrupt are not monitored by the tool.
- **Instruction Set Simulator** This technique treats the compiled programs machine code as its input 'data' and fully simulates the host machine instructions, monitors the code for conditional or unconditional breakpoints or programmer requested "single cycle" animation requests between every step.

Comparison of methods

The advantage of the last method is that no changes are made to the compiled program to provide the diagnostic and there is almost unlimited scope for extensive diagnostics since the tool can augment the host system diagnostics with additional software tracing features. It is also possible to diagnose (and prevent) many program errors automatically using this technique, including storage violations and buffer overflows. Loop detection is also possible using automatic instruction trace together with instruction count thresholds (e.g. pause after 10,000 instructions; display last n instructions) The second method only alters the instruction that will halt before it is executed and may also then restore it before optional resumption by the programmer. Some animators optionally allow the use of more than one method depending on requirements. For example, using method 2 to execute to a particular point at full speed and then using instruction set simulation thereafter.

Additional features

The animator may, or may not, combine other test/debugging features within it such as program trace, dump, conditional breakpoint and memory alteration, program flow alteration, code coverage analysis, "hot spot" detection or similar.

Examples of program animators

- Firebug (Firefox extension)
 - IBM OLIVER (CICS interactive test/debug)
 - SIMON (Batch Interactive test/debug)
-

External links and references

- Stepping (Visual Studio) ^[2] Overview of stepping support in Microsoft Corporation's IDE, Visual Studio
- Tarringim - A Program Animation Environment ^[3]
- Program Animation as a way to teach and learn about Program Design and Analysis ^[4]
- Structured information on software testing (such as the History of Software Testing) published by Testing references ^[5]

References

- [1] See this ad (<http://bdn.borland.com/article/images/20841/tc20ad.jpg>)
- [2] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsdebug/html/_asug_stepping_into_functions.asp
- [3] <http://www.mcs.vuw.ac.nz/comp/Publications/CS-TR-91-2.abs.html>
- [4] http://www.cs.montana.edu/webworks/webworks-home/projects/program_animator/program_animator.html
- [5] <http://www.testingreferences.com/about.php>

Software testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.^[1] Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs (errors or other defects).

Software testing can be stated as the process of validating and verifying that a software program/application/product:

1. meets the requirements that guided its design and development;
2. works as expected; and
3. can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Different software development models will focus the test effort at different points in the development process. Newer development models, such as Agile, often employ test driven development and place an increased portion of the testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been completed.

Overview

Testing can never completely identify all the defects within software.^[2] Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts,^[3] comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. **Software testing** is the process of attempting to make this assessment.

A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy \$59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed.^[4]

History

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979.^[5] Although his attention was on breakage testing ("a successful test is one that finds a bug"^{[5] [6]}) it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:^[7]

- Until 1956 - Debugging oriented^[8]
- 1957–1978 - Demonstration oriented^[9]
- 1979–1982 - Destruction oriented^[10]
- 1983–1987 - Evaluation oriented^[11]
- 1988–2000 - Prevention oriented^[12]

Software testing topics

Scope

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions.^[13] The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.^[14]

Functional vs non-functional testing

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or other performance, behavior under certain constraints, or security. Non-functional requirements tend to be those that reflect the quality of the product, particularly in the context of the suitability perspective of its users.

Defects and failures

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements, that result in errors of omission by the program designer.^[15] A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure.^[16] Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data

or interacting with different software.^[16] A single defect may result in a wide range of failure symptoms.

Finding faults early

It is commonly believed that the earlier a defect is found the cheaper it is to fix it.^[17] The following table shows the cost of fixing the defect depending on the stage it was found.^[18] For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review.

Cost to fix a defect		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Compatibility

A common cause of software failure (real or perceived) is a lack of compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment was capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

Input combinations and preconditions

A very fundamental problem with software testing is that testing under *all* combinations of inputs and preconditions (initial state) is not feasible, even with a simple product.^{[13] [19]} This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, non-functional dimensions of quality (how it is supposed to *be* versus what it is supposed to *do*)—usability, scalability, performance, compatibility, reliability—can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another.

Static vs. dynamic testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

Software verification and validation

Software testing is used in association with verification and validation:^[20]

- Verification: Have we built the software right? (i.e., does it match the specification).
- Validation: Have we built the right software? (i.e., is this what the customer wants).

The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

The software testing team

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing,^[21] different roles have been established: *manager*, *test lead*, *test designer*, *tester*, *automation developer*, and *test administrator*.

Software quality assurance (SQA)

Though controversial, software testing is a part of the software quality assurance (SQA) process.^[13] In SQA, software process specialists and auditors are concerned for the software development process rather than just the artifacts such as documentation, code and systems. They examine and change the software engineering process itself to reduce the amount of faults that end up in the delivered software: the so-called *defect rate*.

What constitutes an "acceptable defect rate" depends on the nature of the software; A flight simulator video game would have much higher defect tolerance than software for an actual airplane.

Although there are close links with SQA, testing departments often exist independently, and there may be no SQA function in some companies.

Software testing is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

Testing methods

The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White box testing

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

- API testing (application programming interface) - testing of the application using public and private APIs

- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - improving the coverage of a test by introducing faults to test code paths
- Mutation testing methods
- Static testing - White box testing includes all static testing

Test coverage

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.^[22]

Two common forms of code coverage are:

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

Black box testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, exploratory testing and specification-based testing.

Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements.^[23] Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing is necessary, but it is insufficient to guard against certain risks.^[24]

Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code *must* have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. On the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion", on the one hand, and the disadvantage of "blind exploring", on the other.^[25]

Grey box testing

Grey box testing (American spelling: **gray box testing**) involves having knowledge of internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by the SWEBOK guide are unit-, integration-, and system testing that are distinguished by the test target without implying a specific process model.^[26] Other test levels are classified by the testing objective.^[27]

Test target

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.^[28]

These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

Unit testing is also called *component testing*.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be localised more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.^[29]

System testing

System testing tests a completely integrated system to verify that it meets its requirements.^[30]

System integration testing

System integration testing verifies that a system is integrated to any external or third-party systems defined in the system requirements.

Objectives of testing

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing is performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.^[31]

Beta testing

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Non-functional testing

Special methods exist to test non-functional aspects of software. In contrast to functional testing, which establishes the correct operation of the software (correct in that it matches the expected behavior defined in the design requirements), non-functional testing verifies that the software functions properly even when it receives invalid or unexpected inputs. Software fault injection, in the form of fuzzing, is an example of non-functional testing. Non-functional testing, especially for software, is designed to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform non-functional testing.

Software performance testing and load testing

Performance testing is executed to determine how fast a system or sub-system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Load testing is primarily concerned with testing that can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*.

Volume testing is a way to test functionality. *Stress testing* is a way to test reliability. *Load testing* is a way to test performance. There is little agreement on what the specific goals of load testing are. The terms load testing, performance testing, reliability testing, and volume testing, are often used interchangeably.

Stability testing

Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of non-functional software testing is often referred to as load (or endurance) testing.

Usability testing

Usability testing is needed to check if the user interface is easy to use and understand. It is concerned mainly with the use of the application.

Security testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

Internationalization and localization

The general ability of software to be internationalized and localized can be automatically tested without actual translation, by using pseudolocalization. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).^[32]

Actual translation to human languages must be tested, too. Possible localization failures include:

- Software is often localized by translating a list of strings out of context, and the translator may choose the wrong translation for an ambiguous source string.
- Technical terminology may become inconsistent if the project is translated by several people without proper coordination or if the translator is imprudent.
- Literal word-for-word translations may sound inappropriate, artificial or too technical in the target language.
- Untranslated messages in the original language may be left hard coded in the source code.
- Some messages may be created automatically at run time and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.
- Software may use a keyboard shortcut which has no function on the source language's keyboard layout, but is used for typing characters in the layout of the target language.
- Software may lack support for the character encoding of the target language.
- Fonts and font sizes which are appropriate in the source language, may be inappropriate in the target language; for example, CJK characters may become unreadable if the font is too small.
- A string in the target language may be longer than the software can handle. This may make the string partly invisible to the user or cause the software to crash or malfunction.
- Software may lack proper support for reading or writing bi-directional text.
- Software may display images with text that wasn't localized.
- Localized operating systems may have differently-named system configuration files and environment variables and different formats for date and currency.

To avoid these and other localization problems, a tester who knows the target language must run the program with all the possible use cases for translation to see if the messages are readable, translated correctly in context and don't cause failures.

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.

The testing process

Traditional CMMI or waterfall development model

A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer.^[33] This practice often results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.^[34]

Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.^[35]

Further information: Capability Maturity Model Integration and Waterfall model

Agile or Extreme development model

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process). The ultimate goal of this test process is to achieve continuous deployment where software updates can be published to the public frequently.^{[36] [37]}

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing.^[38] The sample below is common among organizations employing the Waterfall development model.

- **Requirements analysis:** Testing should begin in the requirements phase of the software development life cycle. During the design phase, testers work with developers in determining what aspects of a design are testable and with what parameters those tests work.
- **Test planning:** Test strategy, test plan, testbed creation. Since many activities will be carried out during testing, a plan is needed.
- **Test development:** Test procedures, test scenarios, test cases, test datasets, test scripts to use in testing software.
- **Test execution:** Testers execute the software based on the plans and test documents then report any errors found to the development team.
- **Test reporting:** Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
- **Test result analysis:** Or Defect Analysis, is done by the development team usually along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with later.
- **Defect Retesting:** Once a defect has been dealt with by the development team, it is retested by the testing team. AKA Resolution testing.
- **Regression testing:** It is common to have a small test program built of a subset of tests, for each integration of new, modified, or fixed software, in order to ensure that the latest delivery has not ruined anything, and that the software product as a whole is still working correctly.

- **Test Closure:** Once the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything that a human can do (and all the ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code including:
 - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
 - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
 - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI
- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into an Integrated Development Environment (IDE).

- A regression testing technique is to have a standard set of tests, which cover existing functionality that result in persistent tabular data, and to compare pre-change data to post-change data, where there should not be differences, using a tool like diffkit. Differences detected indicate unexpected functionality changes or "regression".

Measurement in software testing

Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements as described under the ISO standard ISO/IEC 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of frequently-used software measures, often called *metrics*, which are used to assist in determining the state of the software or the adequacy of the testing.

Testing artifacts

Software testing process can produce several artifacts.

Test plan

A test specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

Traceability matrix

A traceability matrix is a table that correlates requirements or design documents to test documents. It is used to change tests when the source documents are changed, or to verify that the test results are correct.

Test case

A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result.^[39] This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step, or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system, you may also be able to see past test results, who generated the results, and what system configuration was used to generate those results. These past results would usually be stored in a separate table.

Test script

The test script is procedure, or a programming code that replicate the user actions. Initially the term was derived from the product of work created by automated regression test tools. Test Case will be a baseline to create test scripts using a tool or a program.

Test suite

The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Test data

In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project.

Test harness

The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. No certification currently offered actually requires the applicant to demonstrate the ability to test software. No certification is based on a widely accepted body of knowledge. This has led some to declare that the testing field is not ready for certification.^[40] Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.^[41]

Software testing certification types

- *Exam-based*: Formalized exams, which need to be passed; can also be learned by self-study [e.g., for ISTQB or QAI]^[42]
- *Education-based*: Instructor-led sessions, where each course has to be passed [e.g., International Institute for Software Testing (IIST)].

Testing certifications

- Certified Associate in Software Testing (CAST) offered by the QAI^[43]
- CATE offered by the *International Institute for Software Testing*^[44]
- Certified Manager in Software Testing (CMST) offered by the QAI^[43]
- Certified Software Tester (CSTE) offered by the Quality Assurance Institute (QAI)^[43]
- Certified Software Test Professional (CSTP) offered by the *International Institute for Software Testing*^[44]
- CSTP (TM) (Australian Version) offered by *K. J. Ross & Associates*^[45]
- ISEB offered by the Information Systems Examinations Board
- ISTQB Certified Tester, Foundation Level (CTFL) offered by the International Software Testing Qualification Board^{[46] [47]}
- ISTQB Certified Tester, Advanced Level (CTAL) offered by the International Software Testing Qualification Board^{[46] [47]}
- TMPF TMap Next Foundation offered by the *Examination Institute for Information Science*^[48]
- TMPA TMap Next Advanced offered by the *Examination Institute for Information Science*^[48]

Quality assurance certifications

- CMSQ offered by the *Quality Assurance Institute* (QAI)^[43]
- CSQA offered by the *Quality Assurance Institute* (QAI)^[43]
- CSQE offered by the American Society for Quality (ASQ)^[49]
- CQIA offered by the American Society for Quality (ASQ)^[49]

Controversy

Some of the major software testing controversies include:

What constitutes responsible software testing?

Members of the "context-driven" school of testing^[50] believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation.^[51]

Agile vs. traditional

Should testers learn to work under conditions of uncertainty and constant change or should they aim at process "maturity"? The agile testing movement has received growing popularity since 2006 mainly in commercial circles,^{[52] [53]} whereas government and military^[54] software providers use this methodology but also the traditional test-last models (e.g. in the Waterfall model).

Exploratory test vs. scripted^[55]

Should tests be designed at the same time as they are executed or should they be designed beforehand?

Manual testing vs. automated

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly.^[56] More in particular, test-driven development states that developers should write unit-tests of the XUnit type before coding the functionality. The tests then can be considered as a way to capture and implement the requirements.

Software design vs. software implementation^[57]

Should testing be carried out only at the end or throughout the whole process?

Who watches the watchmen?

The idea is that any form of observation is also an interaction—the act of testing can also affect that which is being tested.^[58]

References

- [1] Exploratory Testing (<http://www.kaner.com/pdfs/ETatQAI.pdf>), Cem Kaner, Florida Institute of Technology, *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Orlando, FL, November 2006
- [2] Software Testing (http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/) by Jiantao Pan, Carnegie Mellon University
- [3] Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., "Contract Driven Development = Test Driven Development - Writing Test Cases" (http://se.inf.ethz.ch/people/leitner/publications/cdd_leitner_esec_fse_2007.pdf), Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007, (Dubrovnik, Croatia), September 2007
- [4] Software errors cost U.S. economy \$59.5 billion annually (http://www.abeacha.com/NIST_press_release_bugs_cost.htm), NIST report
- [5] Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. ISBN 0-471-04328-1.
- [6] Company, People's Computer (1987). "Dr. Dobb's journal of software tools for the professional programmer" (<http://books.google.com/?id=7RoIAAAAIAAJ>). *Dr. Dobb's journal of software tools for the professional programmer* (M&T Pub) **12** (1–6): 116. .
- [7] Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
- [8] *until 1956 it was the debugging oriented period, when testing was often associated to debugging: there was no clear difference between testing and debugging*. Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
- [9] *From 1957–1978 there was the demonstration oriented period where debugging and testing was distinguished now - in this period it was shown, that software satisfies the requirements*. Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
- [10] *The time between 1979–1982 is announced as the destruction oriented period, where the goal was to find errors*. Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
- [11] *1983–1987 is classified as the evaluation oriented period: intention here is that during the software lifecycle a product evaluation is provided and measuring quality*. Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
- [12] *From 1988 on it was seen as prevention oriented period where tests were to demonstrate that software satisfies its specification, to detect faults and to prevent faults*. Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
- [13] Kaner, Cem; Falk, Jack and Nguyen, Hung Quoc (1999). *Testing Computer Software, 2nd Ed.*. New York, et al: John Wiley and Sons, Inc.. pp. 480 pages. ISBN 0-471-35846-0.
- [14] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. pp. 41–43. ISBN 0470042125. .
- [15] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. p. 86. ISBN 0470042125. .
- [16] Section 1.1.2, Certified Tester Foundation Level Syllabus (<http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf>), International Software Testing Qualifications Board
- [17] Kaner, Cem; James Bach, Bret Pettichord (2001). *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley. p. 4. ISBN 0-471-08112-4.
- [18] McConnell, Steve (2004). *Code Complete* (2nd ed.). Microsoft Press. pp. 29. ISBN 0-7356-1967-0.
- [19] Principle 2, Section 1.3, Certified Tester Foundation Level Syllabus (<http://www.bcs.org/upload/pdf/istqbsyll.pdf>), International Software Testing Qualifications Board
- [20] Tran, Eushiuan (1999). "Verification/Validation/Certification" (http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html). In Koopman, P.. *Topics in Dependable Embedded Systems*. USA: Carnegie Mellon University. . Retrieved 2008-01-13.
- [21] see D. Gelperin and W.C. Hetzel
- [22] Introduction (<http://www.bullseye.com/coverage.html#intro>), Code Coverage Analysis, Steve Cornett
- [23] Laycock, G. T. (1993) (PostScript). *The Theory and Practice of Specification Based Software Testing* (<http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz>). Dept of Computer Science, Sheffield University, UK. . Retrieved 2008-02-13.

- [24] Bach, James (June 1999). "Risk and Requirements-Based Testing" (http://www.satisfice.com/articles/requirements_based_testing.pdf) (PDF). *Computer* **32** (6): 113–114. . Retrieved 2008-08-19.
- [25] Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 159. ISBN 978-0-615-23372-7.
- [26] <http://www.computer.org/portal/web/swebok/html/ch5#Ref2.1>
- [27] <http://www.computer.org/portal/web/swebok/html/ch5#Ref2.2>
- [28] Binder, Robert V. (1999). *Testing Object-Oriented Systems: Objects, Patterns, and Tools*. Addison-Wesley Professional. p. 45. ISBN 0-201-80938-9.
- [29] Beizer, Boris (1990). *Software Testing Techniques* (Second ed.). New York: Van Nostrand Reinhold. pp. 21,430. ISBN 0-442-20672-0.
- [30] IEEE (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York: IEEE. ISBN 1559370793.
- [31] van Veenendaal, Erik. "Standard glossary of terms used in Software Testing" (<http://www.astqb.org/educational-resources/glossary.php#A>). . Retrieved 17 June 2010.
- [32] Globalization Step-by-Step: The World-Ready Approach to Testing. Microsoft Developer Network (<http://msdn.microsoft.com/en-us/goglobal/bb688148>)
- [33] e)Testing Phase in Software Testing:- (http://www.etestinghub.com/testing_lifecycles.php#2)
- [34] Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. pp. 145–146. ISBN 0-471-04328-1.
- [35] Dustin, Elfriede (2002). *Effective Software Testing*. Addison Wesley. p. 3. ISBN 0-20179-429-2.
- [36] Marchenko, Artem (November 16, 2007). "XP Practice: Continuous Integration" (<http://agilesoftwaredevelopment.com/xp/practices/continuous-integration>). . Retrieved 2009-11-16.
- [37] Gurses, Levent (February 19, 2007). "Agile 101: What is Continuous Integration?" (<http://www.jacoozi.com/blog/?p=18>). . Retrieved 2009-11-16.
- [38] Pan, Jiantao (Spring 1999). "Software Testing (18-849b Dependable Embedded Systems)" (http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/). *Topics in Dependable Embedded Systems*. Electrical and Computer Engineering Department, Carnegie Mellon University. .
- [39] IEEE (1998). *IEEE standard for software test documentation*. New York: IEEE. ISBN 0-7381-1443-X.
- [40] Kaner, Cem (2001). "NSF grant proposal to "lay a foundation for significant improvements in the quality of academic and commercial courses in software testing"" (http://www.testingeducation.org/general/nsf_grant.pdf) (pdf). .
- [41] Kaner, Cem (2003). "Measuring the Effectiveness of Software Testers" (<http://www.testingeducation.org/a/mest.pdf>) (pdf). .
- [42] Black, Rex (December 2008). *Advanced Software Testing- Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager*. Santa Barbara: Rocky Nook Publisher. ISBN 1933952369.
- [43] Quality Assurance Institute (<http://www.qaiglobalinstitute.com/>)
- [44] International Institute for Software Testing (<http://www.testinginstitute.com/>)
- [45] K. J. Ross & Associates (<http://www.kjross.com.au/cstp/>)
- [46] "ISTQB" (<http://www.istqb.org/>). .
- [47] "ISTQB in the U.S." (<http://www.astqb.org/>). .
- [48] EXIN: Examination Institute for Information Science (<http://www.exin-exams.com>)
- [49] American Society for Quality (<http://www.asq.org/>)
- [50] context-driven-testing.com (<http://www.context-driven-testing.com>)
- [51] Article on taking agile traits without the agile method. (<http://www.technicat.com/writing/process.html>)
- [52] "We're all part of the story" (<http://stpcollaborative.com/knowledge/272-were-all-part-of-the-story>) by David Strom, July 1, 2009
- [53] IEEE article about differences in adoption of agile trends between experienced managers vs. young students of the Project Management Institute (<http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/10705/33795/01609838.pdf?temp=x>). See also Agile adoption study from 2007 (<http://www.ambyssoft.com/downloads/surveys/AgileAdoption2007.ppt>)
- [54] Willison, John S. (April 2004). "Agile Software Development for an Agile Force" (<http://web.archive.org/web/20051029135922/http://www.stsc.hill.af.mil/crosstalk/2004/04/0404willison.html>). *CrossTalk* (STSC) (April 2004). Archived from the original (<http://www.stsc.hill.af.mil/crosstalk/2004/04/0404willison.htm>) on unknown. .
- [55] IEEE article on Exploratory vs. Non Exploratory testing (<http://ieeexplore.ieee.org/iel5/10351/32923/01541817.pdf?arnumber=1541817>)
- [56] An example is Mark Fewster, Dorothy Graham: *Software Test Automation*. Addison Wesley, 1999, ISBN 0-201-33140-3.
- [57] Article referring to other links questioning the necessity of unit testing (<http://java.dzone.com/news/why-evangelising-unit-testing->)
- [58] Microsoft Development Network Discussion on exactly this topic (<http://channel9.msdn.com/forums/Coffeehouse/402611-Are-you-a-Test-Driven-Developer/>)

External links

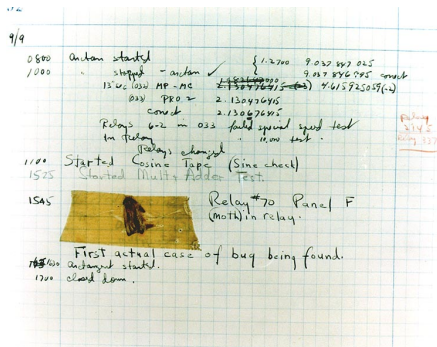
- Software testing tools and products (http://www.dmoz.org/Computers/Programming/Software_Testing/Products_and_Tools/) at the Open Directory Project
- "Software that makes Software better" Economist.com (http://www.economist.com/science/tq/displaystory.cfm?story_id=10789417)
- Automated software testing metrics including manual testing metrics (<http://idtus.com/img/UsefulAutomatedTestingMetrics.pdf>)

Portal:Software Testing

Wikipedia portals: Culture · Geography · Health · History · Mathematics · Natural sciences · People · Philosophy · Religion · Society · Technology

Software testing is the process used to measure the **quality** of developed software.

Keyword-driven testing, also known as **table-driven testing** or **action-word testing**, is a software testing methodology for automated testing that separates the test creation process into two distinct stages: a Planning Stage, and an Implementation Stage. ...More



While Grace Hopper was working on the Harvard Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. Though the term *computer bug* cannot be definitively attributed to Admiral Hopper, she did bring the term into popularity. The remains of the moth can be found in the group's log book at the Smithsonian Institution's National Museum of American History in Washington, D.C..^[1]

Cem Kaner, James D. McCaffrey, James Whittaker & Rex Black

Fuzz testing or '**fuzzing**' is a software testing technique that provides random data ("fuzz") to the inputs of a program. If the program fails, the defects can be noted. The great advantage of fuzz testing is that the test design is extremely simple, and free of preconceptions about system behavior. Fuzz testing is a very simple procedure to implement: Prepare a correct file to input to your program. Replace some part of the file with random data. Open the file with the program. See what breaks.

1. Information technology
2. Software
3. Technology
4. Free software

iOS 4.1: Repeating alarms may trigger incorrectly before or after DST change.^[1]

- Test Automation
- Programming bugs
- Software anomalies
- Software metrics

- Software quality
- Software testing
- Static code analysis
- Risk analysis
 - Abstraction
 - Cohesion
 - Completeness
 - Elegance
 - Extensibility
 - Fault-tolerance
 - Maintainability
 - Reliability
 - Robustness
 - Scalability
 - Testability
- *"Quality is never an accident; it is always the result of intelligent effort."* -- John Ruskin
- *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."* -- Edsger Dijkstra
- *"Beware of bugs in the above code; I have only proved it correct, not tried it."* -- Donald Knuth
- *"Given enough eyeballs, all bugs are shallow."* -- Linus's Law according to Eric S. Raymond
- *"If it ain't broke, you are not trying hard enough."*
- *"Quality is free, but only to those who are willing to pay heavily for it."*
- **Test Process** : Behavior driven development, ISO 9000, ISO 9126, CMM, Static code analysis, Lightweight Software Test Automation, Debugging, Mutation analysis, Equivalence Partitioning, Quality control, Software quality, Software testing, Performance engineering, Formal verification, Risk-based Testing, Fault injection, Fagan inspection, Reliability engineering, Software Quality Assurance, Software inspection, Dynamic program analysis, Symbolic computation, Extreme quality assurance, Test automation, Computerized system validation, Testing Web Sites, Testathon, Quality audit
- **Test levels** : Component or Unit testing, Integration testing, Component integration testing, Acceptance testing, System testing
- **Test types** : Ad hoc testing, Alpha Testing, All-pairs testing, Beta Testing, Black box testing, Boundary testing, Boundary Value Analysis, Build Verification Test, Code coverage, Compatibility testing, Conformance testing, Combinadic, Exploratory testing, Fuzz testing, GUI software testing, Game testing, Hallway testing, Installation testing, Keyword-driven testing, Load testing, Localization testing, Mobile Device Testing, Monkey test, Manual testing, Model-based testing, Playtest, Pseudolocalization, QuickCheck, Regression testing, Recovery testing, Sanity testing, Scenario testing, Soak testing, Software performance testing, Software verification, Smoke testing, Stress testing, Static testing, Session-based testing, Usability testing, White box testing
- **Famous bugs** : List of software bugs
- **People** : Charles E. Brady, Jr., Kenneth D. Cameron, Patrick G. Forrester, Erich Gamma, Charles D. Gemar, Brent Hailpern, Steven Hawley, Cem Kaner, Adam Kolawa, James D. McCaffrey, Brian Marick, Harlan Mills, Stephen S. Oswald, Gene Spafford
- **Companies** : Applabs, AutomatedQA, CTG, Compuware, IBM, Lionbridge, Hewlett Packard HP Software Division, Micro Focus, National Software Testing Laboratories, Segue Software, uTest
- **Test management** : Test strategy, Test Plan, Test effort
- **Tools (commercial)**: AdaTEST95, Automation Anywhere, Cantata++, CAST tool, Coverity, Goanna, IBM OLIVER (CICS interactive test/debug), Insure++, Jinx, Jtest, LDRA Testbed, HP LoadRunner, HP Quality Center, QF-Test, Polyspace, Ranorex, SilkTest, SIMMON, TestComplete, TestPartner, Testware, Time Partition Testing, TOSCA, HP WinRunner
- **Tools (free/open source)**: AutoIt, CfcUnit, CFUnit, Check, Concutest, CPPUnit, Curl-loader, DUnit, Fastest, FindBugs, FitNesse, Framework for Integrated Test, FUnit, HTTP Test Tool, HttpUnit, JMeter, JSystem, JUnit, PHPUnit, Litmus (Mozilla), Mauve (test suite), NUnit, PyUnit, RSpec, Selenium, SimpleTest, soapUI, Splint,

STAF, Watir, WET Web Tester, xUnit

- **Tools (other)** Category:Emulation software, LURCH, Test Automation Framework, Virtual appliance
- **Certification** : British Computer Society, National Software Testing Laboratories, QAI, ISTQB, CSTE
- **Membership associations** : Software Engineering Institute, Association for Software Testing, American Society for Quality
- **Software standards** : IEEE 829, TTCN
- **Terminology** : Software bug, Anomaly in software, Test case, Test suite, Test script, Unusual software bug, System under test, Mock object, Test harness, Test data, Testbed, Test bench, Debugger, Boundary case, Verification and Validation, test plan, Test Anything Protocol, Zarro boogs, Thrash (computer science), Memory debugger, Xqa,
- **Miscellaneous** : Software testing outsourcing, Software metric, List of unit testing frameworks
- Expand the stubs
- Add Citation for verification

What are portals? · List of portals · Featured portals

[2]

References

[1] (<http://support.apple.com/kb/TS3542>).

[2] http://en.wikipedia.org/wiki/Portal%3Asoftware_testing?action=purge

Accelerated stress testing

Accelerated Stress Testing (AST) is an effective method in software engineering, of achieving system robustness by detecting product weaknesses using accelerated stresses, conducting failure analysis, and taking corrective actions. Products may often have hidden defects or weaknesses, which can result in future failures in the field. AST applies stress stimuli to a product to turn such latent defects into observable failures, and therefore offers opportunities to discover and correct product weaknesses early in the product life cycle.

Need for AST

The justification for AST in hardware was originally based on success in practical cases. The need however in AST in the communication and computer manufacturers was driven by the need to achieve high reliability with lower cost and shorter time-to-market. Without the bound to relate with real life stresses, the higher stress levels give higher probability to find the latent defects.

The need for AST may be re-visited nowadays when products often include both software and hardware. Many systems are becoming increasingly complicated so that the number of possible failure modes also increases as newer products are developed.

Types Of Failure Mode

There are 3 types of failure modes were defined: threshold-stress failure, cumulative-stress failure, and combined threshold – cumulative stress failure. Also, the concept of life-time maximum stress for one particular system was introduced there.

- **Threshold-stress failure** is defined as the largest peak stress or combination of stresses, which are encountered by that system throughout its entire product life. Whether the system fails during its product life will therefore depend on whether it can withstand this maximum stress level.
- For **cumulative-stress failure**, we note that time may be included as a stress. The combination of time-stress and other stresses are what manifest the cumulative-stress failure. This combination also has a maximum time-stresses combination over the product life of any system, and therefore also has a threshold value to manifest failure.
- **Combined threshold-cumulative stress failure** is manifested by a threshold-stress first followed by a cumulative-stress, or vice versa.

An example of such a software failure is the overflow of an interim event counter that occurs because a low priority reader process is indefinitely postponed by other high priority processes. Raising the priority of the reader process raises the threshold at which the stress can begin to accumulate. Increasing the size of the event counter increases the amount of continuous stress that can accumulate before the overflow occurs.

Acceptance testing

In engineering and its various subdisciplines, **acceptance testing** is a test conducted to determine if the requirements of a specification or contract are met. It may involve chemical tests, physical tests, or performance tests

In systems engineering it may involve black-box testing performed on a system (for example: a piece of software, lots of manufactured mechanical parts, or batches of chemical products) prior to its delivery.^[1]

It is also known as functional testing, black-box testing, QA testing, application testing, confidence testing, final testing, validation testing, or factory acceptance testing.



Acceptance testing of an aircraft catapult

Software developers often distinguish acceptance testing by the system provider from acceptance testing by the customer (the user or client) prior to accepting transfer of ownership. In the case of software, acceptance testing performed by the customer is known as user acceptance testing (UAT), end-user testing, site (acceptance) testing, or field (acceptance) testing.

A smoke test is used as an acceptance test prior to introducing a build to the main testing process.

Overview

Acceptance testing generally involves running a suite of tests on the completed system. Each individual test, known as a case, exercises a particular operating condition of the user's environment or feature of the system, and will result in a pass or fail, or boolean, outcome. There is generally no degree of success or failure. The test environment is usually designed to be identical, or as close as possible, to the anticipated user's environment, including extremes of such. These test cases must each be accompanied by test case input data or a formal description of the operational activities (or both) to be performed—intended to thoroughly exercise the specific case—and a formal description of the expected results.

Acceptance Tests/Criteria (in Agile Software Development) are usually created by business customers and expressed in a business domain language. These are high-level tests to test the completeness of a user story or stories 'played' during any sprint/iteration. These tests are created ideally through collaboration between business customers, business analysts, testers and developers, however the business customers (product owners) are the primary owners of these tests. As the user stories pass their acceptance criteria, the business owners can be sure of the fact that the developers are progressing in the right direction about how the application was envisaged to work and so it's essential that these tests include both business logic tests as well as UI validation elements (if need be).

Acceptance test cards are ideally created during sprint planning or iteration planning meeting, before development begins so that the developers have a clear idea of what to develop. Sometimes (due to bad planning!) acceptance tests may span multiple stories (that are not implemented in the same sprint) and there are different ways to test them out during actual sprints. One popular technique is to mock external interfaces or data to mimic other stories which might not be played out during an iteration (as those stories may have been relatively lower business priority). A user story is not considered complete until the acceptance tests have passed.

Process

The acceptance test suite is run against the supplied input data or using an acceptance test script to direct the testers. Then the results obtained are compared with the expected results. If there is a correct match for every case, the test suite is said to pass. If not, the system may either be rejected or accepted on conditions previously agreed between the sponsor and the manufacturer.

The objective is to provide confidence that the delivered system meets the business requirements of both sponsors and users. The acceptance phase may also act as the final quality gateway, where any quality defects not previously detected may be uncovered.

A principal purpose of acceptance testing is that, once completed successfully, and provided certain additional (contractually agreed) acceptance criteria are met, the sponsors will then sign off on the system as satisfying the contract (previously agreed between sponsor and manufacturer), and deliver final payment.

User acceptance testing

User Acceptance Testing (UAT) is a process to obtain confirmation that a system meets mutually agreed-upon requirements. A Subject Matter Expert (SME), preferably the owner or client of the object under test, provides such confirmation after trial or review. In software development, UAT is one of the final stages of a project and often occurs before a client or customer accepts the new system.

Users of the system perform these tests, which developers derive from the client's contract or the user requirements specification.

Test-designers draw up formal tests and devise a range of severity levels. Ideally the designer of the user acceptance tests should not be the creator of the formal integration and system test cases for the same system. The UAT acts as a final verification of the required business function and proper functioning of the system, emulating real-world usage conditions on behalf of the paying client or a specific large customer. If the software works as intended and without

issues during normal use, one can reasonably extrapolate the same level of stability in production.

User tests, which are usually performed by clients or end-users, do not normally focus on identifying simple problems such as spelling errors and cosmetic problems, nor showstopper defects, such as software crashes; testers and developers previously identify and fix these issues during earlier unit testing, integration testing, and system testing phases.

The results of these tests give confidence to the clients as to how the system will perform in production. There may also be legal or contractual requirements for acceptance of the system.

Q-UAT - Quantified User Acceptance Testing

Quantified User Acceptance Testing (Q-UAT or, more simply, the "Quantified Approach") is a revised Business Acceptance Testing process which aims to provide a smarter and faster alternative to the traditional UAT phase. Depth-testing is carried out against business requirements only at specific planned points in the application or service under test. A reliance on better quality code-delivery from the development/build phase is assumed and a complete understanding of the appropriate business process is a pre-requisite. This methodology - if carried out correctly - results in a quick turnaround against plan, a decreased number of test scenarios which are more complex and wider in breadth than traditional UAT and ultimately the equivalent confidence-level attained via a shorter delivery-window, allowing products/changes to come to market quicker.

The Q-UAT approach depends on a "gated" three-dimensional model. The key concepts are:

1. Linear Testing (LT, the 1st dimension)
2. Recursive Testing (RT, the 2nd dimension)
3. Adaptive Testing (AT, the 3rd dimension).

The four "gates" which conjoin and support the 3-dimensional model act as quality safeguards and include contemporary testing concepts such as:

- Internal Consistency Checks (ICS)
- Major Systems/Services Checks (MSC)
- Realtime/Reactive Regression (RTR).

The Quantified Approach was shaped by the former "guerilla" method of acceptance testing which was itself a response to testing phases which proved too costly to be sustainable for many small/medium-scale projects.

Acceptance testing in Extreme Programming

Acceptance testing is a term used in agile software development methodologies, particularly Extreme Programming, referring to the functional testing of a user story by the software development team during the implementation phase.

The customer specifies scenarios to test when a user story has been correctly implemented. A story can have one or many acceptance tests, whatever it takes to ensure the functionality works. Acceptance tests are black box system tests. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release. A user story is not considered complete until it has passed its acceptance tests. This means that new acceptance tests must be created for each iteration or the development team will report zero progress.^[2]

Types of acceptance testing

Typical types of acceptance testing include the following

User acceptance testing

This may include factory acceptance testing, i.e. the testing done by factory users before the factory is moved to its own site, after which site acceptance testing may be performed by the users at the site.

Operational Acceptance Testing (OAT)

Also known as operational readiness testing, this refers to the checking done to a system to ensure that processes and procedures are in place to allow the system to be used and maintained. This may include checks done to back-up facilities, procedures for disaster recovery, training for end users, maintenance procedures, and security procedures.

Contract and regulation acceptance testing

In contract acceptance testing, a system is tested against acceptance criteria as documented in a contract, before the system is accepted. In regulation acceptance testing, a system is tested to ensure it meets governmental, legal and safety standards.

Alpha and beta testing

Alpha testing takes place at developers' sites, and involves testing of the operational system by internal staff, before it is released to external customers. Beta testing takes place at customers' sites, and involves testing by a group of customers who use the system at their own locations and provide feedback, before the system is released to other customers. The latter is often called "field testing".

List of development to production (testing) environments

- **DEV**, Development Environment [1]
- **DTE**, Development Testing Environment
- **QA**, Quality Assurance (Testing Environment) [2]
- **DIT**, Development Integration Testing
- **DST**, Development System Testing
- **SIT**, System Integration Testing
- **UAT**, User Acceptance Testing [3]
- **PROD**, Production Environment [4]

[1-4] Usual development environment stages in medium-sized development projects.

List of acceptance-testing frameworks

- FitNesse, a fork of Fit
 - Framework for Integrated Test (*Fit*)
 - iMacros
 - ItsNat Java Ajax web framework with built-in, server based, functional web testing capabilities.
 - Ranorex
 - Selenium (software)
 - Test Automation FX
 - Watir
-

References

- [1] Black, Rex (August 2009). *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. Hoboken, NJ: Wiley. ISBN 0-470-40415-9.
- [2] Acceptance Tests (<http://www.extremeprogramming.org/rules/functionaltests.html>)

External links

- Acceptance Test Engineering Guide (<http://testingguidance.codeplex.com>) by Microsoft patterns & practices (<http://msdn.com/practices>)
- Article Using Customer Tests to Drive Development (<http://www.methodsandtools.com/archive/archive.php?id=23>) from Methods & Tools (<http://www.methodsandtools.com/>)
- Article Acceptance TDD Explained (<http://www.methodsandtools.com/archive/archive.php?id=72>) from Methods & Tools (<http://www.methodsandtools.com/>)

Ad hoc testing

Ad hoc testing is a commonly used term for software testing performed without planning and documentation (but can be applied to early scientific experimental studies).

The tests are intended to be run only once, unless a defect is discovered. Ad hoc testing the least formal test method. As such, it has been criticized because it is not structured and hence defects found using this method may be harder to reproduce (since there are no written test cases). However, the strength of ad hoc testing is that important defects can be found quickly.

It is performed by improvisation: the tester seeks to find bugs by any means that seem appropriate. Ad hoc testing can be seen as a light version of error guessing, which itself is a light version of exploratory testing.

References

- Exploratory Testing Explained ^[1]
- Context-Driven School of testing ^[2]

References

- [1] <http://www.satisfice.com/articles/et-article.pdf>
 - [2] <http://www.context-driven-testing.com/>
-

Agile testing

Agile testing is a software testing practice that follows the principles of agile software development. Agile testing does not emphasize testing procedures and focuses on ongoing testing against newly developed code until quality software from an end customer's perspective results. Agile testing is built upon the philosophy that testers need to adapt to rapid deployment cycles and changes in testing patterns.

Overview

Agile testing involves testing from the customer perspective as early as possible, testing early and often as code becomes available and stable enough, since working increments of the software are released often in agile software development. This is commonly done by using automated acceptance testing to minimize the amount of manual labor involved.

Further reading

- Lisa Crispin, Janet Gregory (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley. ISBN 0-321-53446-8.
- Ambler, Scott (2010). "Agile Testing and Quality Strategies: Discipline over Rhetoric" ^[1]. Retrieved 2010-07-15.
- Kalistick (2011). "Leading Agile testing" ^[2]. Retrieved 2011-07-11.

References

- Pettichord, Bret. "Agile Testing What is it? Can it work?" ^[3]. Retrieved 2008-12-11.
- Hendrickson, Elisabeth (2008-08-11). "Agile Testing, Nine Principles and Six Concrete Practices for Testing on Agile Teams" ^[4]. Retrieved 2011-04-26.
- Parkinson, Shane (2008-11-26). "Agile Methodology" ^[5]. Retrieved 2008-11-26.
- Egan, Patrick (2008-12-15). "Video: How does agile affect testing" ^[6]. Retrieved 2008-11-26.
- Crispin, Lisa (2003-03-21). "XP Testing Without XP: Taking Advantage of Agile Testing Practices" ^[7]. Retrieved 2009-06-11.
- Lerche-Jensen, Steen (2003-10-18). "Agile Certifications - Agile Testing" ^[8]. Retrieved 2010-10-18.

Agile testing conference

- "Agile Testing Days Conference - Europe" ^[9]. 2010-10-04. Retrieved 2009-10-04.

References

- [1] <http://www.ambysoft.com/essays/agileTesting.html>
- [2] <http://www.kalistick.com/agile-testing-helps-testing-efficiency.html>
- [3] http://www.io.com/~wazmo/papers/agile_testing_20021015.pdf
- [4] <http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf>
- [5] <http://agiletesting.com.au/agile-methodology/agile-methods-and-software-testing/>
- [6] http://www.agilejournal.com/component/option,com_seyret/Itemid,0/task,videodirectlink/id,49/
- [7] <http://www.methodsandtools.com/archive/archive.php?id=2>
- [8] <http://www.waqb.org>
- [9] <http://www.agiletestingdays.com>

All-pairs testing

All-pairs testing or **pairwise testing** is a combinatorial software testing method that, for each pair of input parameters to a system (typically, a software algorithm), tests all possible discrete combinations of those parameters. Using carefully chosen test vectors, this can be done much faster than an exhaustive search of all combinations of all parameters, by "parallelizing" the tests of parameter pairs. The number of tests is typically $O(nm)$, where n and m are the number of possibilities for each of the two parameters with the most choices.

The reasoning behind all-pairs testing is this: the simplest bugs in a program are generally triggered by a single input parameter. The next simplest category of bugs consists of those dependent on interactions between pairs of parameters, which can be caught with all-pairs testing.^[1] Bugs involving interactions between three or more parameters are progressively less common,^[2] whilst at the same time being progressively more expensive to find by exhaustive testing, which has as its limit the exhaustive testing of all possible inputs.^[3]

Many testing methods regard all-pairs testing of a system or subsystem as a reasonable cost-benefit compromise between often computationally infeasible higher-order combinatorial testing methods, and less exhaustive methods which fail to exercise all possible pairs of parameters. Because no testing technique can find all bugs, all-pairs testing is typically used together with other quality assurance techniques such as unit testing, symbolic execution, fuzz testing, and code review.

Notes

- [1] Black, Rex (2007). *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. New York: Wiley. p. 240. ISBN 978-0-470-12790-2.
- [2] D.R. Kuhn, D.R. Wallace, A.J. Gallo, Jr. (June 2004). "Software Fault Interactions and Implications for Software Testing" (<http://csrc.nist.gov/groups/SNS/acts/documents/TSE-0172-1003-1.pdf>). *IEEE Trans. on Software Engineering* **30** (6). .
- [3] (2010) Practical Combinatorial Testing. SP 800-142. (<http://csrc.nist.gov/groups/SNS/acts/documents/SP800-142-101006.pdf>). Natl. Inst. of Standards and Technology. (Report).

External links

- Combinatorialtesting.com; Includes clearly written introductions to pairwise and other, more thorough, methods of combinatorial testing (<http://www.combinatorialtesting.com>)
- Hexawise.com - Pairwise test case generating tool with both free and commercial versions (also provides more thorough 3-way, 4-way, 5-way, and 6-way coverage solutions) (<http://hexawise.com/>)
- Pairwise Testing Comes of Age - Review including history, examples, issues, research (<http://testcover.com/pub/background/stareast2008.ppt>)
- Pairwise Testing: Combinatorial Test Case Generation (<http://www.pairwise.org/>)
- Pairwise testing (<http://www.developsense.com/testing/PairwiseTesting.html>)
- All-pairs testing (<http://www.mcdowella.demon.co.uk/allPairs.html>)
- Pairwise and generalized t-way combinatorial testing (<http://csrc.nist.gov/acts/>)
- TestApi - the API library for testing, providing a variation generation API (<http://testapi.codeplex.com>)

American Software Testing Qualifications Board

The **American Software Testing Qualifications Board (ASTQB)** was founded in 2003 as the American Testing Board. In April 2005, the name was changed to the American Software Testing Qualifications Board.

ASTQB is a non-profit organization whose members comprise a group of highly experienced experts in software testing who volunteer their time to the development, maintenance, and promotion of the ISTQB Certified Tester program in the U.S. They also represent U.S. interests internationally as the national board for the U.S. within the International Software Testing Qualifications Board (ISTQB). The ISTQB is responsible for the international qualification scheme called "ISTQB Certified Tester".

ISTQB certificates at Foundation Level are also dual certified by ISEB, which was established in 1967 and is the world's leading issuer of Software Testing certifications and the only certifications aligned with the worldwide Professionalism in IT campaign.

ASTQB's exam and accreditation fees are charged to cover the cost connected with the administration of exams, applications for accreditation, the maintenance of a physical office, exhibits at leading software testing conventions, and the employment of administrative staff.

References

- International Software Testing Qualifications Board (ISTQB)
- Verify Conference ^[1]
- Acronym Finder ^[2]

External links

- ASTQB Official Website ^[3]
- About ASTQB ^[4]
- ISTQB Official Website ^[5]
- ISEB Official Website ^[6]

References

- [1] http://verifyconference.com/index2.php?option=com_content&do_pdf=1&id=68
- [2] <http://www.acronymfinder.com/acronym.aspx?rec={B7906901-D965-4C84-9BF7-4D2339C64605}>
- [3] <http://www.astqb.org>
- [4] <http://www.astqb.org/displaycommon.cfm?an=1&subarticlenbr=17>
- [5] <http://www.istqb.org>
- [6] <http://www.iseb-exams.com>
-

API Sanity Autotest

API Sanity Autotest

Developer(s)	linuxtesting.org
Initial release	November 30, 2009
Stable release	1.12.5 / June 3, 2011
Written in	Perl
Operating system	Linux, FreeBSD, Mac OS X, MS Windows
License	GPLv2, LGPLv2
Website	ispras.linuxfoundation.org ^[1]

API Sanity Autotest (ASAT) is a unit test generator for shared libraries written in C and C++ programming languages. The main feature of this framework is the ability to completely automatically generate reasonable (in most, but unfortunately not all, cases) input parameters for every function from the library API. This allows to quickly cover any C/C++ library API by "shallow"-quality tests and catch serious problems like crashes or program hanging.

The tool was developed by the Russian Linux Verification Center at the Institute for System Programming of the Russian Academy of Sciences (ISPRAS).

External Links

- Home Page ^[2]
- 2011 GSoC LSB projects: CUnit format support for API Sanity Autotest ^[3]
- 2010 GSoC LSB projects: Annotation support for API Sanity Autotest ^[4]
- API Sanity Autotest at The FreeBSD Fresh Ports ^[5]
- API Sanity Autotest at A Survey and Classification of Software Testing Tools ^[6]

References

- [1] http://ispras.linuxfoundation.org/index.php/API_Sanity_Autotest
- [2] http://ispras.linux-foundation.org/index.php/API_Sanity_Autotest
- [3] <http://www.linuxfoundation.org/collaborate/workgroups/gsoc/2011-gsoc-lsb-projects>
- [4] <http://www.linuxfoundation.org/collaborate/workgroups/gsoc/2010-gsoc-lsb-projects>
- [5] <http://www.freshports.org/devel/api-sanity-autotest/>
- [6] <http://www.doria.fi/handle/10024/63006>

Association for Software Testing

The **Association for Software Testing**, commonly referred to as the **AST**, is dedicated to advancing the understanding of the science and practice of software testing according to context-driven principles ^[1]. **AST**'s membership consists of scholars, students and practitioners who are interested in the advancement of the field of software testing. The group was founded in the United States in 2004.

AST has multiple objectives including:

- Fostering cross-pollination of ideas between scholars, students and practitioners.
- Hosting an annual conference focused on cross-community information sharing.
- Promoting ethical behavior for all software testers.
- Supporting peer workshops related to software testing.

Conference

The AST's first conference, named "CAST" for the Conference of the Association for Software Testing, was held in Indianapolis, Indiana in 2006 and had the theme "Influencing the Practice".

CAST 2007 had a theme of "Testing Techniques: Innovations and Applications" and was held in Bellevue, Washington, USA

CAST 2008 had a theme of "Beyond the Boundaries: Interdisciplinary Approaches to Software Testing" and was held in Toronto, Ontario, Canada

CAST 2009 had a theme of "Serving Our Stakeholders" and was held in Colorado Springs, Colorado

CAST 2010 had a theme of "Skills in Testing" and was held in Grand Rapids, Michigan.

CAST 2011 has a theme of "Context-Driven Testing" and will be held in Seattle, Washington.

Training

The AST offers a series of online training courses in black box software testing (BBST), based on videos from Florida Institute of Technology's Center for Software Testing Education & Research (CSTER) with additional study aids and support from live instructors.

The initial set of courses enhances materials developed under a series of grants from the National Science Foundation. These materials are used in traditional university courses and in courses for practitioners, such as those offered by AST. The AST courses run 4 weeks each and focus on a single topic or test technique. AST is planning new courses by additional instructors.

External links

- AST official website ^[2]
- Articles of Incorporation ^[3]PDF (70.7 KiB)
- Cem Kaner, Rebecca L. Fiedler, & Scott Barber, "Building a free courseware community around an online software testing curriculum." MERLOT conference, Minneapolis, August 2008. ^[4]

References

[1] <http://www.context-driven-testing.com>

[2] <http://www.associationforsoftwaretesting.org/>

[3] <http://www.associationforsoftwaretesting.org/wp-content/uploads/ArticlesOfIncorporation04152004.pdf>

[4] http://conference.merlot.org/2008/Saturday/kaner_c_Saturday.pdf

Attack patterns

In computer science, **attack patterns** are a group of rigorous methods for finding bugs or errors in code related to computer security.

Attack patterns are often used for testing purposes and are very important for ensuring that potential vulnerabilities are prevented. The attack patterns themselves can be used to highlight areas which need to be considered for security hardening in a software application. They also provide, either physically or in reference, the common solution pattern for preventing the attack. Such a practice can be termed *defensive coding patterns*.

Attack patterns define a series of repeatable steps that can be applied to simulate an attack against the security of a system.

Categories

There are several different ways to categorize attack patterns. One way is to group them into general categories, such as: Architectural, Physical, and External (see details below). Another way of categorizing attack patterns is to group them by a specific technology or type of technology (e.g. database attack patterns, web application attack patterns, network attack patterns, etc. or SQL Server attack patterns, Oracle Attack Patterns, .Net attack patterns, Java attack patterns, etc.)

Using General Categories

Architectural attack patterns are used to attack flaws in the architectural design of the system. These are things like weaknesses in protocols, authentication strategies, and system modularization. These are more logic-based attacks than actual bit-manipulation attacks.

Physical attack patterns are targeted at the code itself. These are things such as SQL injection attacks, buffer overflows, race conditions, and some of the more common forms of attacks that have become popular in the news.

External attack patterns include attacks such as trojan horse attacks, viruses, and worms. These are not generally solvable by software-design approaches, because they operate relatively independently from the attacked program. However, vulnerabilities in a piece of software can lead to these attacks being successful on a system running the vulnerable code. An example of this is the vulnerable edition of Microsoft SQL Server, which allowed the Slammer worm to propagate itself.^[1] The approach taken to these attacks is generally to revise the vulnerable code.

Structure

Attack Patterns are structured very much like structure of Design patterns. Using this format is helpful for standardizing the development of attack patterns and ensures that certain information about each pattern is always documented the same way.

A recommended structure for recording Attack Patterns is as follows:

- **Pattern Name**

The label given to the pattern which is commonly used to refer to the pattern in question.

- **Type & Subtypes**

The pattern type and its associated subtypes aid in classification of the pattern. This allows users to rapidly locate and identify pattern groups that they will have to deal with in their security efforts.

Each pattern will have a type, and zero or more subtypes that identify the category of the attack pattern. Typical types include Injection Attack, Denial of Service Attack, Cryptanalysis Attack, etc. Examples of typical subtypes for Denial Of Service for example would be: DOS – Resource Starvation, DOS-System Crash, DOS-Policy Abuse.

Another important use of this field is to ensure that true patterns are not repeated unnecessarily. Often it is easy to confuse a new exploit with a new attack. New exploits are created all the time for the same attack patterns. The Buffer Overflow Attack Pattern is a good example. There are many known exploits, and viruses that take advantage of a Buffer Overflow vulnerability. But they all follow the same pattern. Therefore the Type and Subtype classification mechanism provides a way to classify a pattern. If the pattern you are creating doesn't have a unique Type and Subtype, chances are it's a new exploit for an existing pattern.

This section is also used to indicate if it is possible to automate the attack. If it is possible to automate the attack, it is recommended to provide a sample in the Sample Attack Code section which is described below.

- **Also Known As**

Certain attacks may be known by several different names. This field is used to list those other names.

- **Description**

This is a description of the attack itself, and where it may have originated from. It is essentially a free-form field that can be used to record information that doesn't easily fit into the other fields.

- **Attacker Intent**

This field identifies the intended result of the attacker. This indicates the attacker's main target and goal for the attack itself. For example, The Attacker Intent of a DOS – Bandwidth Starvation attack is to make the target web site unreachable to legitimate traffic.

- **Motivation**

This field records the attacker's reason for attempting this attack. It may be to crash a system in order to cause financial harm to the organization, or it may be to execute the theft of critical data in order to create financial gain for the attacker.

This field is slightly different than the Attacker Intent field in that it describes why the attacker may want to achieve the Intent listed in the Attacker Intent field, rather than the physical result of the attack.

- **Exploitable Vulnerability**

This field indicates the specific or type of vulnerability that creates the attack opportunity in the first place. An example of this in an Integer Overflow attack would be that the integer based input field is not checking size of the value of the incoming data to ensure that the target variable is capable of managing the incoming value. This is the vulnerability that the associated exploit will take advantage of in order to carry out the attack.

- **Participants**

The Participants are one or more entities that are required for this attack to succeed. This includes the victim systems as well as the attacker and the attacker's tools or system components. The name of the entity should be accompanied by a brief description of their role in the attack and how they interact with each other.

- **Process Diagram**

These are one or more diagrams of the attack to visually explain how the attack is executed. This diagram can take whatever form is appropriate but it is recommended that the diagram be similar to a system or class diagram showing data flows and the components involved.

- **Dependencies and Conditions**

Every attack must have some context to operate in and the conditions that make the attack possible. This section describes what conditions are required and what other systems or situations need to be in place in order for the attack to succeed. For example, for the attacker to be able to execute an Integer Overflow attack, they must have access to the vulnerable application. That will be common amongst most of the attacks. However if the vulnerability only exposes itself when the target is running on a remote RPC server, that would also be a condition that would be noted here.

- **Sample Attack Code**

If it is possible to demonstrate the exploit code, this section provides a location to store the demonstration code. In some cases, such as a Denial of Service attack, specific code may not be possible. However in Overflow, and Cross Site Scripting type attacks, sample code would be very useful.

- **Existing Exploits**

Exploits can be automated or manual. Automated exploits are often found as viruses, worms and hacking tools. If there are any existing exploits known for the attack this section should be used to list a reference to those exploits. These references can be internal such as corporate knowledge bases, or external such as the various CERT, and Virus databases.

Exploits are not to be confused with vulnerabilities. An Exploit is an automated or manual attack that utilises the vulnerability. It is not a listing of a vulnerability found in a particular product for example.

- **Follow-On Attacks**

Follow-on attacks are any other attacks that may be enabled by this particular attack pattern. For example, a Buffer Overflow attack pattern, is usually followed by Escalation of Privilege attacks, Subversion attacks or setting up for Trojan Horse / Backdoor attacks. This field can be particularly useful when researching an attack and identifying what other potential attacks may have been carried out or set up.

- **Mitigation Types**

The mitigation types are the basic types of mitigation strategies that would be used to prevent the attack pattern. This would commonly refer to Security Patterns and Defensive Coding Patterns. Mitigation Types can also be used as a means of classifying various attack patterns. By classifying Attack Patterns in this manner, libraries can be developed to implement particular mitigation types which can then be used to mitigate entire classes of Attack Patterns. This libraries can then be used and reused throughout various applications to ensure consistent and reliable coverage against particular types of attacks.

- **Recommended Mitigation**

Since this is an attack pattern, the recommended mitigation for the attack can be listed here in brief. Ideally this will point the user to a more thorough mitigation pattern for this class of attack.

- **Related Patterns**

This section will have a few subsections such as Related Patterns, Mitigation Patterns, Security Patterns, and Architectural Patterns. These are references to patterns that can support, relate to or mitigate the attack and the listing for the related pattern should note that.

An example of related patterns for an Integer Overflow Attack Pattern is:

Mitigation Patterns – Filtered Input Pattern, Self Defending Properties pattern

Related Patterns – Buffer Overflow Pattern

- **Related Alerts, Listings and Publications**

This section lists all the references to related alerts listings and publications such as listings in the Common Vulnerabilities and Exposures list, CERT, SANS, and any related vendor alerts. These listings should be hyperlinked to the online alerts and listings in order to ensure it references the most up to date information possible.

- CVE: [2]
- CWE: [3]
- CERT: [4]

Various Vendor Notification Sites.

Further reading

- Alexander, Christopher; Ishikawa, Sara; & Silverstein, Murray. *A Pattern Language*. New York, NY: Oxford University Press, 1977
- Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software* ISBN 0201633612, Addison-Wesley, 1995
- Thompson, Herbert; Chase, Scott, *The Software Vulnerability Guide* ISBN 1584503580, Charles River Media, 2005
- Gegick, Michael & Williams, Laurie. “*Matching Attack Patterns to Security Vulnerabilities in Software-Intensive System Designs.*” ACM SIGSOFT Software Engineering Notes, Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications SESS '05, Volume 30, Issue 4, ACM Press, 2005
- Howard, M.; & LeBlanc, D. *Writing Secure Code* ISBN 0735617228, Microsoft Press, 2002.
- Moore, A. P.; Ellison, R. J.; & Linger, R. C. *Attack Modeling for Information Security and Survivability*, Software Engineering Institute, Carnegie Mellon University, 2001
- Hogg, Greg & McGraw, Gary. *Exploiting Software: How to Break Code* ISBN 0201786958, Addison-Wesley, 2004
- McGraw, Gary. *Software Security: Building Security In* ISBN 0321356705, Addison-Wesley, 2006
- Viega, John & McGraw, Gary. *Building Secure Software: How to Avoid Security Problems the Right Way* ISBN 020172152X, Addison-Wesley, 2001
- Schumacher, Markus; Fernandez-Buglioni, Eduardo; Hybertson, Duane; Buschmann, Frank; Sommerlad, Peter *Security Patterns* ISBN 0470858842, John Wiley & Sons, 2006
- Koizol, Jack; Litchfield, D.; Aitel, D.; Anley, C.; Eren, S.; Mehta, N.; & Riley, H. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes* ISBN 0764544683, Wiley, 2004
- Schneier, Bruce. *Attack Trees: Modeling Security Threats* Dr. Dobbs's Journal, December, 1999

References

- [1] PSS Security Response Team Alert - New Worm: W32.Slammer (<http://www.microsoft.com/technet/security/alerts/slammer.mspx>)
- [2] <http://cve.mitre.org/>
- [3] <http://cwe.mitre.org/>
- [4] http://www.cert.org/nav/index_red.html
- fuzzdb: (<http://fuzzdb.googlecode.com>)

Augmented Reality-based testing

Augmented Reality-based testing (ARBT) is a test method that combines Augmented Reality and Software testing to enhance testing by inserting an additional dimension into the testers field of view. For example, a tester wearing a Head Mounted Display (HMD) or Augmented reality contact lenses ^[1] that places images of both the physical world and registered virtual graphical objects over the user's view of the world can detect virtual labels on areas of a system to clarify test operating instructions for a tester who is performing tests on a complex system.

In 2009 as a spin-off to Augmented Reality for Maintenance and Repair (ARMAR) ^[2] Alexander Andelkovic coined the idea 'Augmented Reality-based testing' introducing the idea of using Augmented Reality together with software testing.

Overview

The test environment of technology is becoming more complex, this puts higher demand on test engineers to have higher knowledge, testing skills and work effective. A powerful unexplored dimension that can be utilized is the Virtual environment, a lot of information and data that today is available but unpractical to use due to overhead in time needed to gather and present can with ARBT be used instantly.

Application

ARBT can be of help in following test environments:

Support

Assembling and disassembling a test object ^[3] can be learned out and practice scenarios can be run through to learn how to fix fault scenarios that may occur.

Guidance

Minimizing risk of misunderstanding complex test procedures can be done by virtually describing test steps in front of the tester on the actual test object.

Educational

Background information about test scenario with earlier bugs found pointed out on the test object and reminders to avoid repeating previous mistakes made during testing of selected test area.

Training

Junior testers can learn complex test scenarios with less supervision. Test steps will be pointed out and information about pass criteria need to be confirmed the junior tester can train before the functionality is finished and do some regression testing.

Informational

Tester can point at a physical object and get detailed updated technical data and information needed to perform selected test task.

Inspire

Testers performing exploratory testing that need inspiration of areas to explore can get instant information about earlier exploratory test sessions gathered through Session-based testing.

References

- [1] Babak A. Parviz, Augmented Reality in a Contact Lens (<http://spectrum.ieee.org/biomedical/bionics/augmented-reality-in-a-contact-lens/>) IEEE Spectrum inside technology biomedical bionics, Sep 2009.
- [2] Steve Henderson, Steven Feiner. ARMAR: Augmented Reality for Maintenance and Repair (ARMAR) (<http://graphics.cs.columbia.edu/projects/armar/index.htm>). Columbia University Computer Graphics & User Interfaces Lab.
- [3] BMW research labs, Augmented reality BMW car repair (<http://www.youtube.com/watch?v=P9KPJIA5yds>) Video Clip, Oct 2007.

Australian and New Zealand Testing Board

Australia and New Zealand Testing Board

Type	Professional organization
Founded	September 2005
Key people	Chris Carter (Chair) David Fuller (Board Member and Web Master) Sharon Robson (Chair of Marketing) Graeme MacKenzie (Chair of Examination Panel) Josephine Crawford (Chair of the Accreditation Panel) Steve Toms (Treasurer) Ian Ross (Board Member) David Hayman (Vice Chair Panel)
Area served	Australia and New Zealand
Method	Certification, industry standards, conferences
Volunteers	~14
Employees	1
Members	Over 2000
Motto	"The ANZTB offers sought after certification, dependable training accreditation and career-enhancing support for software testing professionals throughout Australia and New Zealand."
Website	Official website ^[1]

The **Australia and New Zealand Testing Board** (ANZTB) is a non-profit professional association with the purpose of supporting and promoting the discipline of software testing. It was founded in Sydney, Australia in September 2005 when it was admitted by the International Software Testing Board (ISTQB) as one of their National Boards.

The activities of ANZTB include:

- Development and management of examinations against the ISTQB International Syllabus that the industry can rely upon.
- Accreditation of Training Providers
- Annual conference in Software Testing
- Free local Special Interest Groups for Software Testers
- Support for Software Testing Professionals
- Promotion of Software Testing as a Career choice.

The ANZTB is responsible for the preparing and managing examinations for the international qualification scheme called "ISTQB Certified Tester". Examinations are held around Australia and New Zealand several times per year. The qualifications are based on a syllabus, and there is a hierarchy of qualifications and guidelines for accreditation and examination.

The ANZTB offers the following examinations:

- (1) The ISTQB Foundation Level exam
- (2) The ISTQB Advanced Test Manager
- (3) The ISTQB Advanced Test Analyst
- (4) The ISTQB Advanced Technical Test Analyst will be available soon.

The contents of each syllabus are taught as courses by ANZTB Accredited training providers. They are globally marketed under the brand name "ISTQB Certified Tester".

After the examination, each successful participant receives the "ISTQB/ANZTB Certified Tester" Certificate.

References

[1] <http://www.anztb.org>

External links

- ANZTB Official Website (<http://www.anztb.org/>)
- ISTQB Official Website (<http://www.istqb.org>)
- ISTQB Syllabus Foundation Level 2007 (<http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf>)
- ISTQB Syllabus Advanced Level 2007 (http://www.istqb.org/downloads/syllabi/CTAL_Syllabus_V_2007.pdf)
- ISTQB Standard Glossary of Terms used in Software Testing V.2.0, Dec, 2nd 2007 (<http://www.istqb.org/downloads/glossary-current.pdf>)

Automated Testing Framework

ATF or **Automated Testing Framework** is a testing framework originally created for NetBSD as a Google Summer of Code project in 2007.^[1] **Automated Testing Framework** is also used in many mobile phone companies to test latest applications or updated OS. **ATF** is a very useful tool which does many basic and time-consuming works such as clicking and switching applications repeatedly for developers. In addition, daily regression test will increase chance to catch bugs before the release of new features.

ATF is a software testing framework in which test cases can be written in JAVA, POSIX shell, C, or C++.

A primary goal of the ATF project is that tests are self-contained and intended to be executed by end users periodically.

It is released under the two-clause BSD license.

[1] Automated Testing Framework: About (<http://www.netbsd.org/~jmmv/atf/about.html>)

Avalanche (dynamic analysis tool)

For other uses, see Avalanche (disambiguation)

Avalanche is a dynamic program analysis tool developed in ISP RAS that performs symbolic execution in order to generate input data that causes an analysed program to crash. Avalanche uses dynamic binary instrumentation framework provided by Valgrind to collect a set of constraints which are then solved by STP^[1] constraint solver.

Avalanche is open source.

External links

- [Avalanche Homepage](#)^[2]

References

[1] <http://sites.google.com/site/stpfastprover/>

[2] <http://code.google.com/p/avalanche/>

User:Awright415/Centercode

Centercode

Type	Private
Founded	2001
Headquarters	Laguna Hills, California
Industry	Software
Products	Centercode Connect, OnlineBeta.com, Managed Beta Tests
Slogan	
Website	Centercode.com ^[1]

Centercode is a software and professional services company which offers beta test management solutions. Centercode's offerings are the Centercode Connect SaaS based beta test management platform and Managed Beta Testing professional services. Centercode was founded in 2001.

Products and services

Centercode Connect

Centercode Connect is a web-based based platform for managing beta tests, including:

- **Community Building** - Tools to build and maintain a beta customer community
- **User Management** - User monitoring and management with automation capabilities
- **Recruitment** - Targeted beta tester recruitment
- **Content** - Text and file based content (beta project news, tutorials, etc.)
- **Releases** - Secure beta build download, product key distribution, and release information
- **Digital Agreements** - Management of confidentiality agreements (NDAs)
- **Discussion Forums** - Moderated beta tester communication and support
- **Feedback** - Feedback and work-flow engine for bug reports, suggestions, etc.
- **Surveys** - Custom beta tester questionnaires
- **Tasks** - Activities for beta tester direction and regression testing
- **Wikis** - Collaborative resource building (e.g. product manuals, support materials)
- **Reporting** - Custom reports for feedback and activities

Managed Beta Tests

Centercode provides outsourced managed beta tests. These include the following phases:

- Project plan design
 - Beta product distribution
 - Beta participant recruiting and selection
 - Beta participation management
 - Closure reports
 - Beta product collection
 - Incentive program
-

OnlineBeta.com Beta Tester Communities

Centercode's OnlineBeta.com is a free service which provides individuals world-wide with the opportunity to participate in beta tests. Members of this community are invited to apply for Centercode managed beta tests. Members may also opt-in for notifications of new beta testing communities built using Centercode Connect, offering additional opportunities to participate in beta tests.

Competition

- VocOnline (formerly BetaSphere)
- Prefinery
- Customer Feedback Solutions

External links

- Centercode ^[2]
- Centercode Blog ^[3]
- OnlineBeta.com ^[4]

References

[1] <http://www.centercode.com/>

[2] <http://www.centercode.com>

[3] <http://www.centercode.com/blog/>

[4] <http://www.onlinebeta.com>

Bebugging

Bebugging (or **fault seeding**) is a popular software engineering technique used in the 1970s to measure test coverage. Known bugs are randomly added to a program source code and the programmer is tasked to find them. The percentage of the known bugs not found gives an indication of the real bugs that remain.

The earliest application of bebugging was Harlan Mills's fault seeding approach ^[1] which was later refined by stratified fault-seeding ^[2]. These techniques worked by adding a number of known faults to a software system for the purpose of monitoring the rate of detection and removal. This assumed that it is possible to estimate the number of remaining faults in a software system still to be detected by a particular test methodology.

Bebugging is a type of fault injection.

References

[1] H. D. Mills, "On the Statistical Validation of Computer Programs," IBM Federal Systems Division 1972.

[2] L. J. Morell and J. M. Voas, "Infection and Propagation Analysis: A Fault-Based Approach to Estimating Software Reliability," College of William and Mary in Virginia, Department of Computer Science September, 1988.

Behavior Driven Development

Behavior driven development (or **BDD**) is an agile software development technique that encourages collaboration between developers, QA and non-technical or business participants in a software project. It was originally named in 2003 by Dan North^[1] as a response to Test Driven Development, including Acceptance Test or Customer Test Driven Development practices as found in Extreme Programming. It has evolved over the last few years^[2].

On the "Agile Specifications, BDD and Testing eXchange" in November 2009 in London, Dan North^[3] gave the following definition of BDD:

BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.

BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders. It extends TDD by writing test cases in a natural language that non-programmers can read. Behavior-driven developers use their native language in combination with the *ubiquitous language* of domain driven design to describe the purpose and benefit of their code. This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc.

Dan North created the first ever BDD framework, JBehave^[1], followed by a story-level BDD framework for Ruby called RBehave^[4] which was later integrated into the RSpec project^[5]. He also worked with David Chelimsky, Aslak Hellestøy and others to develop RSpec and also to write "The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends". The first story-based framework in RSpec was later replaced by Cucumber mainly developed by Aslak Hellestøy.

In 2008, Chris Matts, who was involved in the first discussions around BDD, came up with the idea of Feature Injection, allowing BDD to cover the analysis space and provide a full treatment of the software lifecycle from vision through to code and release.

BDD practices

The practices of BDD include:

- Establishing the goals of different stakeholders required for a vision to be implemented
 - Drawing out features which will achieve those goals using feature injection
 - Involving stakeholders in the implementation process through outside-in software development
 - Using examples to describe the behavior of the application, or of units of code
 - Automating those examples to provide quick feedback and regression testing
 - Using 'should' when describing the behavior of software to help clarify responsibility and allow the software's functionality to be questioned
 - Using 'ensure' when describing responsibilities of software to differentiate outcomes in the scope of the code in question from side-effects of other elements of code.
 - Using mocks to stand-in for collaborating modules of code which have not yet been written
-

Outside-in

BDD is driven by business value^[6]; that is, the benefit to the business which accrues once the application is in production. The only way in which this benefit can be realized is through the user interface(s) to the application, usually (but not always) a GUI.

In the same way, each piece of code, starting with the UI, can be considered a stakeholder of the other modules of code which it uses. Each element of code provides some aspect of behavior which, in collaboration with the other elements, provides the application behavior.

The first piece of production code that BDD developers implement is the UI. Developers can then benefit from quick feedback as to whether the UI looks and behaves appropriately. Through code, and using principles of good design and refactoring, developers discover collaborators of the UI, and of every unit of code thereafter. This helps them adhere to the principle of YAGNI, since each piece of production code is required either by the business, or by another piece of code already written.

Application examples in the Gherkin language

The requirements of a retail application might be, "Refunded or exchanged items should be returned to stock."

In BDD, a developer or QA engineer might clarify the requirements by breaking this down into specific examples, e.g.

Note: The language of the examples below is called **Gherkin** and is used in **cucumber for ruby**^[7], **specflow for dotnet**^[8] and **behat for php**^[9]

Scenario 1: Refunded items should be returned to stock

- **Given** a customer previously bought a black sweater from me
- **and** I currently have three black sweaters left in stock
- **when** he returns the sweater for a refund
- **then** I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock

- **Given** that a customer buys a blue garment
- **and** I have two blue garments in stock
- **and** three black garments in stock.
- **When** he returns the garment for a replacement in black,
- **Then** I should have three blue garments in stock
- **and** two black garments in stock

Each scenario is an exemplar, designed to illustrate a specific aspect of behavior of the application.

When discussing the scenarios, participants question whether the outcomes described always result from those events occurring in the given context. This can help to uncover further scenarios which clarify the requirements^[10]. For instance, a domain expert noticing that refunded items are not always returned to stock might reword the requirements as "Refunded or replaced items should be returned to stock unless faulty."

This in turn helps participants to pin down the scope of requirements, which leads to better estimates of how long those requirements will take to implement.

The words Given, When and Then are often used to help drive out the scenarios, but are not mandated.

These scenarios can also be automated, if an appropriate tool exists to allow automation at the UI level. If no such tool exists then it may be possible to automate at the next level in, i.e.: if an MVC design pattern has been used, the level of the Controller.

Programmer-domain examples and behavior

The same principles of examples, using contexts, events and outcomes are used to drive development at the level of abstraction of the programmer, as opposed to the business level. For instance, the following examples describe an aspect of behavior of a list:

Example 1: New lists are empty

- **Given** a new list
- **Then** the list should be empty.

Example 2: Lists with things in them are not empty.

- **Given** a new list
- **When** we add an object
- **Then** the list should not be empty.

Both these examples are required to describe the behavior of the

```
list.isEmpty()
```

method, and to derive the benefit of the method. These examples are usually automated using TDD frameworks. In BDD these examples are often encapsulated in a single method, with the name of the method being a complete description of the behavior. Both examples are required for the code to be valuable, and encapsulating them in this way makes it easy to question, remove or change the behavior.

For instance, using Java and JUnit 4, the above examples might become:

```
public class ListTest {

    @Test
    public void shouldKnowWhetherItIsEmpty() {
        List list1 = new List();
        assertTrue(list1.isEmpty());

        List list2 = new List();
        list2.add(new Object());
        assertFalse(list2.isEmpty());
    }
}
```

Other practitioners, particularly in the Ruby community, prefer to split these into two separate examples, based on separate contexts for when the list is empty or has items in. This technique is based on Dave Astels' practice, "One assertion per test"^[11].

Sometimes the difference between the context, events and outcomes is made more explicit. For instance:

```
public class WindowControlBehavior {

    @Test
    public void shouldCloseWindows() {

        // Given
        WindowControl control = new WindowControl("My AFrame");
        AFrame frame = new AFrame();
```

```
// When
control.closeWindow();

// Then
ensureThat(!frame.isShowing());
}
}
```

However the example is phrased, the effect describes the behavior of the code in question. For instance, from the examples above one can derive:

- List should know when it is empty
- WindowControl should close windows

The description is intended to be useful if the test fails, and to provide documentation of the code's behavior. Once the examples have been written they are then run and the code implemented to make them work in the same way as TDD. The examples then become part of the suite of regression tests.

Using mocks

BDD proponents claim that the use of "should" and "ensureThat" in BDD examples encourages developers to question whether the responsibilities they're assigning to their classes are appropriate, or whether they can be delegated or moved to another class entirely. Practitioners use an object which is simpler than the collaborating code, and provides the same interface but more predictable behavior. This is injected into the code which needs it, and examples of that code's behavior are written using this object instead of the production version.

These objects can either be created by hand, or created using a mocking framework such as Mockito, Moq, NMock, Rhino Mocks, JMock or EasyMock.

Questioning responsibilities in this way, and using mocks to fulfill the required roles of collaborating classes, encourages the use of Role-based Interfaces. It also helps to keep the classes small and loosely coupled.

References

- [1] D.North, Introducing Behaviour Driven Development (<http://dannorth.net/introducing-bdd>)
- [2] D.North, comments, The RSpec Book - Question about Chapter 11: Writing software that matters (<http://forums.pragprog.com/forums/95/topics/3035>)
- [3] Dan North: How to sell BDD to the business (<http://skillsmatter.com/podcast/java-jee/how-to-sell-bdd-to-the-business>)
- [4] D.North, Introducing RBehave (<http://dannorth.net/2007/06/introducing-rbehave>)
- [5] S.Miller, InfoQ: RSpec incorporates RBehave (<http://www.infoq.com/news/2007/10/RSpec-incorporates-RBehave>)
- [6] E.Keogh, BDD - TDD done well? (<http://lizkeogh.com/2007/06/13/bdd-tdd-done-well/>)
- [7] cucumber for ruby (<https://github.com/cucumber/cucumber/wiki/Gherkin>)
- [8] specflow for dotnet (<https://github.com/techtalk/SpecFlow/wiki/Using-Gherkin-Language-in-SpecFlow>)
- [9] behat for php (<http://docs.behat.org/guides/1.gherkin.html>)
- [10] D.North, What's in a Story (<http://dannorth.net/whats-in-a-story>)
- [11] D. Astels, One assertion per test (<http://techblog.daveastels.com/tag/bdd/>)

External links

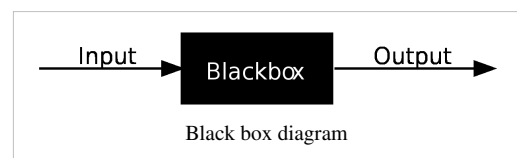
- Dan North's article introducing BDD (<http://dannorth.net/introducing-bdd>)
- Introduction to Behavior Driven Development (<http://behavior-driven.org/>)
- Say Hello To Behavior Driven Development (BDD)- Part 1 (http://www.codeproject.com/KB/architecture/Say_Hello_To_BDD.aspx)
- Say Hello To Behavior Driven Development (BDD)- Part 2 (http://www.codeproject.com/KB/architecture/Say_Hello_World_To_BDD.aspx)
- Behavior Driven Development Using Ruby (Part 1) (<http://www.oreillynet.com/pub/a/ruby/2007/08/09/behavior-driven-development-using-ruby-part-1.html>)
- Behavior-Driven Development Using Ruby (Part 2) (<http://www.oreillynet.com/pub/a/ruby/2007/08/30/behavior-driven-development-using-ruby-part-2.html>)
- In pursuit of code quality: Adventures in behavior-driven development by Andrew Glover (<http://www.ibm.com/developerworks/java/library/j-cq09187/index.html>)
- The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends (<http://www.pragprog.com/titles/achbd/the-rspec-book>)

Black-box testing

Black-box testing is a method of software testing that tests the functionality of an application as opposed to its internal structures or workings (see white-box testing). Specific knowledge of the application's code/internal structure and programming knowledge in general is not required. Test cases are built around

specifications and requirements, i.e., what the application is supposed to do. It uses external descriptions of the software, including specifications, requirements, and designs to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure.

This method of test can be applied to all levels of software testing: unit, integration, functional, system and acceptance. It typically comprises most if not all testing at higher levels, but can also dominate unit testing as well.



Test design techniques

Typical black-box test design techniques include:

- Decision table testing
- All-pairs testing
- State transition tables
- Equivalence partitioning
- Boundary value analysis.

Boundary value analysis:

- i) Elements at the edge of the domain are selected and tested.
- ii) Instead of focusing on input condition only, the test cases from output domain are also derived.
- iii) Test case design technique that complements equivalence partitioning technique. by Shanavas R [MCA].

In this approach, the domain of a program is partitioned into a set of equivalence classes. The partitioning is done such that the behaviour of the program is similar to every input data belonging to the same equivalence class.

Hacking

In penetration testing, black-box testing refers to a methodology where an ethical hacker has no knowledge of the system being attacked. The goal of a black-box penetration test is to simulate an external hacking or cyber warfare attack.

External links

- BCS SIGIST (British Computer Society Specialist Interest Group in Software Testing): *Standard for Software Component Testing* ([http://www.testingstandards.co.uk/Component Testing.pdf](http://www.testingstandards.co.uk/Component%20Testing.pdf)), Working Draft 3.4, 27. April 2001.

Block design

In combinatorial mathematics, a **block design** is a particular kind of hypergraph or set system, which has applications to experimental design, finite geometry, software testing, cryptography, and algebraic geometry. Many variations have been studied, including **balanced incomplete block designs**.^{[1] [2]}

Given a finite set X (of elements called points) and integers $k, r, \lambda \geq 1$, we define a *2-design* B to be a set of k -element subsets of X , called *blocks*, such that the number r of blocks containing x in X is independent of x , and the number λ of blocks containing given distinct points x and y in X is also independent of the choices.

Here v (the number of elements of X , called points), b (the number of blocks), k, r , and λ are the *parameters* of the design. (Also, B may not consist of all k -element subsets of X ; that is the meaning of *incomplete*.) In a table:

v	points, number of elements of X
b	blocks
r	number of blocks containing a given point
k	number of points in a block
λ	number of blocks containing 2 (or more generally t) points

The design is called a (v, k, λ) -design or a (v, b, r, k, λ) -design. The parameters are not all independent; v, k , and λ determine b and r , and not all combinations of v, k , and λ are possible. The two basic equations connecting these parameters are

$$bk = vr,$$

$$\lambda(v - 1) = r(k - 1).$$

These conditions are not sufficient as for example a $(43,7,1)$ -design does not exist. A fundamental theorem, Fisher's inequality, named after Ronald Fisher, is that $b \geq v$ in any block design. The case of equality is called a symmetric design; it has many special features.

Examples

Examples of block designs include the lines in finite projective planes (where X is the set of points of the plane and $\lambda = 1$), and Steiner triple systems ($k = 3$ and $\lambda = 1$). The former is a relatively simple example of a symmetric design. Triple systems ($k = 3$) are of interest in their own right.^[3]

Projective planes

Projective planes are a special case of block designs, where we have $v > 0$ points and, as they are symmetric designs, $b = v$ (which is the limit case of Fisher's inequality), from the first basic equation we get

$$k = r,$$

and since $\lambda = 1$ by definition, the second equation gives us

$$v - 1 = k(k - 1).$$

Now, given an integer $n \geq 1$, called the *order of the projective plane*, we can put $k = n + 1$ and, from the displayed equation above, we have $v = (n + 1)n + 1 = n^2 + n + 1$ points in a projective plane of order n .

Since a projective plane is symmetric, we have that $b = v$, which means that $b = n^2 + n + 1$ also. The number b is usually called the number of *lines* of the projective plane.

This means, as a corollary, that in a projective plane, the number of lines and the number of points are always the same. For a projective plane, k is the number of points on each line and it is equal to $n + 1$, where n is the order of the plane. Similarly, $r = n + 1$ is the number of lines to which the a given point is incident.

For $n = 2$ we get a projective plane of order 2, also called the Fano plane, with $v = 4 + 2 + 1 = 7$ points and 7 lines. In the Fano plane, each line has $n + 1 = 3$ points and each point belongs to $n + 1 = 3$ lines.

Biplane geometry

A **biplane geometry** or **biplane** is another type of symmetric design ("projective design"), here with $\lambda = 2$ – it is a symmetric design such that any set of two points is contained in two blocks ("lines"), while any two lines intersect in two points;^[4] this is similar to a finite projective plane, except that rather than two points determining one line (and two lines determining one point), they determine two lines (respectively, points). A biplane geometry of order n is one whose blocks have $k = n + 2$ points, by analogy with a projective plane of order n being one with $k = n + 1$ points, and similarly for other projective designs. A biplane of order n has $v = 1 + (k + 2)(k + 1)/2$ points (since $r = k$).

As examples:^[5]

- The order 0 biplane has 2 points (and lines of size 2 – (2,2,2)); it is two points, with two blocks, each consisting of both points. Geometrically, it is the digon.

One can also in define trivial biplane geometries of order -1 (1 point, lines of size 1 (2,1,2) – the point is contained in the line) and -2 (1 point, lines of size 0 (2,0,2) – the point is not contained in the line).

- The order 1 biplane has 4 points (and lines of size 3 – (4,3,2)); it is the complete design with $v = 4$ and $k = 3$. Geometrically, the points are the vertices and the blocks are the faces of a tetrahedron.
- The order 2 biplane is the complement of the Fano plane: it has 7 points (and lines of size 4 – (7,4,2)), where the lines are given as the *complements* of the (3-point) lines in the Fano plane.^[6]
- The order 3 biplane has 11 points (and lines of size 5 – (11,5,2)), and is also known as the **Paley biplane** after Raymond Paley; it is associated to the Paley digraph of order 11, which is constructed using the field with 11 elements, and is associated to the order 12 Hadamard matrix; see Paley construction I.

Algebraically this corresponds to the exceptional embedding of the projective special linear group $PSL(2, 5)$ in $PSL(2, 11)$ – see projective linear group: action on p points for details.^[6]

- There are three biplanes of order 4 (16 points, lines of size 6 – (16,6,2)).

- There are five biplanes of order 9 (and 56 points, lines of size 11 – (56,11,2)).^[7]

Generalization: t -designs

Given any integer $t \geq 2$, a t -design B is a class of k -element subsets of X , called *blocks*, such that every point x in X appears in exactly r blocks, and every t -element subset T appears in exactly λ blocks. The numbers v (the number of elements of X), b (the number of blocks), k , r , λ , and t are the *parameters* of the design. The design may be called a t -(v,k,λ)-design. Again, these four numbers determine b and r and the four numbers themselves cannot be chosen arbitrarily. The equations are

$$b_i = \lambda \binom{v-i}{t-i} / \binom{k-i}{t-i} \text{ for } i = 0, 1, \dots, t,$$

where b_i is the number of blocks that contain any i -element set of points.

There are no known examples of non-trivial t -($v,k,1$)-designs with $t > 5$.

The term *block design* by itself usually means a 2-design.

Notes

- [1] Handbook of combinatorial designs. Edited by Charles J. Colbourn and Jeffrey H. Dinitz. Second edition. Discrete Mathematics and its Applications (Boca Raton). Chapman & Hall/CRC, Boca Raton, FL, 2007
- [2] Stinson, Douglas R. Combinatorial designs: Constructions and analysis. Springer-Verlag, New York, 2004. xvi+300 pp. ISBN 0-387-95487-2
- [3] Colbourn, Charles J. and Rosa, Alexander, Triple systems, Oxford Mathematical Monographs, The Clarendon Press Oxford University Press, New York. 1999, ISBN 0-19-853576-7
- [4] *ATLAS of Finite Groups*, p. 7
- [5] Designs and their codes, by E. F. Assmus, J. D. Key, p. 126 (<http://books.google.com/books?id=OE5IHAZuKUQC&pg=PA126>)
- [6] Martin, Pablo; Singerman, David (April 17, 2008), *From Biplanes to the Klein quartic and the Buckyball* (<http://www.neverendingbooks.org/DATA/biplanesingerman.pdf>), p. 4.
- [7] Kaski, Petteri and Östergård, Patric (2008). "There Are Exactly Five Biplanes with $k = 11$ ". *Journal of Combinatorial Designs* **16** (2): 117–127. doi:10.1002/jcd.20145. MR2008m:05038.

References

- van Lint, J.H., and R.M. Wilson (1992), *A Course in Combinatorics*. Cambridge, Eng.: Cambridge University Press.
- S. S. Shrikhande, and Vasanti N. Bhat-Nayak, Non-isomorphic solutions of some balanced incomplete block designs I – *Journal of Combinatorial Theory*, 1970
- Raghavarao, Damaraju (1988). *Constructions and Combinatorial Problems in Design of Experiments* (corrected reprint of the 1971 Wiley ed.). New York: Dover.
- Raghavarao, Damaraju and Padgett, L.V. (2005). *Block Designs: Analysis, Combinatorics and Applications*. World Scientific.
- Street, Anne Penfold and Street, Deborah J. (1987). *Combinatorics of Experimental Design*. Oxford U. P. [Clarendon]. pp. 400+xiv. ISBN 0198532563.
- Weisstein, Eric W., "Block Designs (<http://mathworld.wolfram.com/BlockDesign.html>)" from MathWorld.

External links

- Design DB (<http://batman.cs.dal.ca/~peter/designdb/>): A database of combinatorial, statistical, experimental block designs

Boundary case

The term **boundary case** is frequently used in software engineering to refer to the behavior of a system when one of its inputs is at or just beyond its maximum or minimum limits. It is frequently used when discussing software testing.

For example, if an input field is meant to accept only integer values 0 - 100, entering the values -1, 0, 100, and 101 would represent the boundary cases.

It is commonly thought that three cases should be used when boundary testing (one on the boundary, and one on either side to it). However, the case on the valid side of the boundary is redundant, and so equivalence partitioning recommends skipping it..

Boundary testing

Boundary testing or boundary value analysis, is where test cases are generated using the **extremes of the input domain**, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values. It is similar to Equivalence Partitioning but focuses on "corner cases".^[1] ^[2]

References

- [1] Software Testing and Quality Assurance Glossary (<http://www.aptest.com/glossary.html#bvatesting>)
- [2] Video Tutorial on YouTube (<http://www.youtube.com/watch?v=wNMM8Bc0G3A>)

Boundary-value analysis

Boundary value analysis is a software testing technique in which tests are designed to include representatives of boundary values. Values on the minimum and maximum edges of an equivalence partition are tested. The values could be either input or output ranges of a software component. Since these boundaries are common locations for errors that result in software faults they are frequently exercised in test cases.

Application

The expected input and output values to the software component should be extracted from the component specification. The values are then grouped into sets with identifiable boundaries. Each set, or partition, contains values that are expected to be processed by the component in the same way. Partitioning of test data ranges is explained in the equivalence partitioning test case design technique. It is important to consider both valid and invalid partitions when designing test cases.

For an example, if the input values were months of the year, expressed as integers, the input parameter 'month' might have the following partitions:

```

... -2 -1  0 1 ..... 12 13 14 15 .....
-----|-----|-----
invalid partition 1   valid partition   invalid partition 2

```

The boundary between two partitions is the place where the behavior of the application changes and is not a real number itself. The boundary value is the minimum (or maximum) value that is at the boundary. The number 0 is the maximum number in the first partition, the number 1 is the minimum value in the second partition, both are boundary values. Test cases should be created to generate inputs or outputs that will fall on and to either side of each boundary, which results in two cases per boundary. The test cases on each side of a boundary should be in the

smallest increment possible for the component under test, for an integer this is 1, but the input was a decimal with 2 places then it would be .01. In the example above there are boundary values at 0,1 and 12,13 and each should be tested.

Boundary value analysis does not require invalid partitions. Take an example where a heater is turned on if the temperature is 10 degrees or colder. There are two partitions (temperature \leq 10, temperature $>$ 10) and two boundary values to be tested (temperature=10, temperature=11).

Where a boundary value falls within the invalid partition the test case is designed to ensure the software component handles the value in a controlled manner. Boundary value analysis can be used throughout the testing cycle and is equally applicable at all testing phases.

References

- The Testing Standards Working Party ^[1] website.

References

[1] <http://www.testingstandards.co.uk>

Browser speed test

A browser speed test is a computer benchmark to measure the performance of the JavaScript engine of a web browser. In general the software is available online, located on a website, where different algorithms are loaded and performed in the browser client. Typical test tasks are rendering and animation, DOM transformations, string operations, mathematical calculations, sorting algorithms and memory instructions. Browser speed tests have been used during browser wars to prove superiority of specific web browsers. The popular Acid3 test is no particular speed test but checks browser conformity to web standards (though it checks whether a general performance goal is met or not).

General tests

Peacekeeper

Online speed test by Futuremark, mainly using rendering, mathematical and memory operations. Takes approx. 5 minutes for execution and tells results of other browsers with different CPUs. Does not respect operating system.

Speed-Battle

Test of JavaScript engine using simple algorithms. Displays results of other visitors (best, average, poorest) with same operating system and browser version. Additional statistics page with browser ranking.

Developer Suites

SunSpider

The test suite is a component of the WebKit rendering engine by Apple. As WebKit is used by Google Chrome and Safari, these browsers achieve good results with SunSpider (2-3 times faster than Firefox). Internet Explorer used to be very slow on this test but from Internet Explorer 9 it has been the fastest ^[1]. The test approx. 2 minutes for execution and does not test rendering performance.

V8

JavaScript test suite by Google, used to optimize Google Chrome web browser. Chrome achieves about five times better results than Firefox 4 and Opera, and about two and half times better results than Safari with this benchmark (tested on Mac OS X). Does not test rendering performance.

Dromaeo

Mozilla test suite based on SunSpider tests. Takes several minutes for execution and displays very detailed information about every single test task.

Criticism

All browser tests deliver different results depending on the type and structure of testing and the focus of their measurement, though the tendency of results seems always to be the same. JavaScript and JavaScript engine speed in general are not the only criteria to evaluate the speed of a browser. Loading and rendering speed for a specific website via the Internet, memory consumption, hard disk storage consumption, start-up speed and so on should also be considered to rate the performance of a browser but are normally not included in online browser speed tests.

References

[1] Preston Gralla (2011-03-16). "Internet Explorer 9 speeds past the competition" (http://www.computerworld.com/s/article/9214674/Internet_Explorer_9_speeds_past_the_competition). Computer World. . Retrieved 2011-03-21.

External links

- Peacekeeper (<http://service.futuremark.com/peacekeeper/index.action>)
- Speed-Battle (<http://www.speed-battle.com>)
- V8 benchmark (<http://v8.googlecode.com/svn/data/benchmarks/v6/run.html>)
- SunSpider (<http://www2.webkit.org/perf/sunspider-0.9/sunspider.html>)
- Dromaeo (<http://dromaeo.com>)
- Kraken (<http://krakenbenchmark.mozilla.com/>)
- Are We Fast Yet? (<http://arewefastyet.com/>) - A comparison of v8 and SunSpider benchmarks on various browsers.

BS 7925-1

BS 7925-1 is BSI's standard glossary of software testing terms. This standard complements its partner BS 7925-2 (software component testing).

BS 7925-1 was developed by the Testing Standards Working Party,^[1] sponsored by BCS SIGiST, and published in August 1998.

References

[1] "Testing Standards Working Party" (<http://www.testingstandards.co.uk>). . Retrieved 3 July 2010.

External links

- BS 7925-1 (http://www.testingstandards.co.uk/bs_7925-1.htm) at the Testing Standards website
- BSI Group (<http://www.bsigroup.co.uk>)
- British Computer Society Specialist Interest Group in Software Testing (<http://www.sigist.org.uk/>)

BS 7925-2

BS 7925-2 is BSI's software component testing standard.^[1]

The standard was developed by the Testing Standards Working Party,^[2] sponsored by BCS SIGiST, and published in August 1998.

References

[1] "BS 7925-2:1998" (<http://shop.bsigroup.com/en/ProductDetail/?pid=000000000001448026>). BSI Group. . Retrieved 3 July 2010.

[2] "Testing Standards Working Party" (<http://www.testingstandards.co.uk>). . Retrieved 3 July 2010.

External links

- BS 7925-2 (http://www.testingstandards.co.uk/bs_7925-2.htm) at the Testing Standards website
 - BSI Group (<http://www.bsigroup.co.uk>)
 - British Computer Society Specialist Interest Group in Software Testing (<http://www.sigist.org.uk/>)
-

Bug bash

In software development, a **bug bash** is where all the developers, testers, program managers, usability researchers, designers, documentation folks, and even sometimes marketing people, put aside their regular day-to-day duties and pound on the product to get as many eyes on the product as possible.^[1]

Bug bash sounds similar to eat one's own dog food and is a tool used as part of test management approach. Bug bash is usually declared in advance to the team. The test management team sends out the scope and assigns the testers as resource to assist in setup and also collect bugs. Test management might use this along with small token prize for good bugs found and/or have small socials (drinks) at the end of the Bug Bash. Another interesting bug bash prize was to Pieing test management team members.

References

[1] Ron Patton (2001). *Software Testing*. Sams. ISBN 0672319837.

Build verification test

In software testing, a **Build Verification Test (BVT)**, also known as Build Acceptance Test, is a set of tests run on each new build of a product to verify that the build is testable before the build is released into the hands of the test team. The build acceptance test is generally a short set of tests, which exercises the mainstream functionality of the application software. Any build that fails the build verification test is rejected, and testing continues on the previous build (provided there has been at least one build that has passed the acceptance test).

BVT is important because it lets developers know right away if there is a serious problem with the build, and they save the test team wasted time and frustration by avoiding test of an unstable build.

CA/EZTEST

CA/EZTEST was a CICS interactive test/debug software package distributed by Computer Associates and originally called EZTEST/CICS, produced by Capex Corporation of Phoenix, Arizona with assistance from Ken Dakin from England.

The product provided source level test and debugging features for programs written in COBOL, PL/1 and Assembler languages to complement their own existing COBOL optimizer product.

Competition

CA/EZTEST initially competed with two rival products:

- "Interrest" originally from On-line Software International, based in the US and
 - OLIVER (CICS interactive test/debug) from Advanced Programming Techniques in the UK
- and, much later, in the early 1990s
- XPEDITER from Compuware Corporation who in 1994 acquired the OLIVER product.^[1]

Eventually, CA, Inc. purchased Interrest from On-line software , renamed it CA-INTERTEST, and stopped selling CA/EZTEST [2].

Early critical role

Between them, these three products provided much needed third party system software support for IBM's "flagship" teleprocessing product CICS, which survived for more than 20 years as a strategic product without any memory protection of its own. A single "rogue" application program (frequently by a buffer overflow) could accidentally overwrite data almost anywhere in the address space causing "down-time" for the entire teleprocessing system, possibly supporting thousands of remote terminals. This was despite the fact that much of the world's banking and other commerce relied heavily on CICS for secure transaction processing between 1970 and early 1990s. The difficulty in deciding which application program caused the problem was often insurmountable and frequently the system would be restarted without spending many hours investigated very large (and initially unformatted) "core dump"s requiring expert system programming support and knowledge.

Early integrated testing environment

Additionally, the product (and its competitors) provided an integrated testing environment which was not provided by IBM for early versions of CICS and which was only partially satisfied with their later embedded testing tool — "Execution Diagnostic Facility" (EDF), which only helped newer "Command level" programmers and provided no protection.

Supported operating systems

The following operating systems were supported:

- IBM MVS
 - IBM XA
 - IBM VSE (except XPEDITER)
-

References

- [1] Compuware Corp (<http://www.answers.com/topic/compuware-corporation>), Answers.com.
- [2] http://en.wikipedia.org/wiki/CA_Technologies

External links

- IBM CICS official website (<http://www.ibm.com/cics>)
- Xpediter — Interactive mainframe analysis and debugging (<http://www.compuware.com/solutions/xpediter.asp>)
- Xpeditor/CICS users guide for COBOL (<http://middleware.its.state.nc.us/middleware/CICS/Documentation/Xpeditor/CWXCUC7I.pdf>) for OS/390 (Release 2.5 or above) and z/OS, September 2004
- CA Inc. — product description for CA-Interrest (<http://www3.ca.com/solutions/ProductFamily.aspx?ID=1320>)

Cause-effect graph

In software testing, a cause-effect graph is a directed graph that maps a set of causes to a set of effects. The causes may be thought of as the input to the program, and the effects may be thought of as the output. Usually the graph shows the nodes representing the causes on the left side and the nodes representing the effects on the right side. There may be intermediate nodes in between that combine inputs using logical operators such as AND and OR.

Constraints may be added to the causes and effects. These are represented as edges labelled with the constraint symbol using a dashed line. For causes, valid constraint symbols are E (exclusive), O (one and only one), and I (at least one). The exclusive constraint states that both causes1 and cause2 cannot be true simultaneously. The Inclusive (at least one) constraint states that at least one of the causes 1, 2 or 3 must be true. The OaOO (One and Only One) constraint states that only one of the causes 1, 2 or 3 can be true.

For effects, valid constraint symbols are R (Requires) and M (Mask). The Requires constraint states that if cause 1 is true, then cause 2 must be true, and it is impossible for 1 to be true and 2 to be false. The mask constraint states that if effect 1 is true then effect 2 is false. (Note that the mask constraint relates to the effects and not the causes like the other constraints.

The graph's direction is as follows:

```
Causes --> intermediate nodes --> Effects
```

The graph can always be rearranged so there is only one node between any input and any output. See conjunctive normal form and disjunctive normal form.

A cause-effect graph is useful for generating a reduced decision table.

Further reading

- Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley & Sons. ISBN 0471043281.

Characterization test

In computer programming, a **characterization test** is a means to describe (characterize) the **actual** behaviour of an existing piece of software, and therefore protect existing behaviour of legacy code against unintended changes via automated testing. This term was coined by Michael Feathers ^[1]

The goal of characterization tests is to help developers verify that the modifications made to a reference version of a software system did not modify its behaviour in unwanted or undesirable ways. They enable, and provide a safety net for, extending and refactoring code that does not have adequate unit tests.

When creating a characterization test, one must observe what outputs occur for a given set of inputs. Given an observation that the legacy code gives a certain output based on given inputs, then a test can be written that asserts that the output of the legacy code matches the observed result for the given inputs. For example, if one observes that `f(3.14) == 42`, then this could be created as a characterization test. Then, after modifications to the system, the test can determine if the modifications caused changes in the results when given the same inputs.

Unfortunately, as with any testing, it is generally not possible to create a characterization test for every possible input and output. As such, many people opt for either statement or branch coverage. However, even this can be difficult. Test writers must use their judgment to decide how much testing is appropriate. It is often sufficient to write characterization tests that only cover the specific inputs and outputs that are known to occur, paying special attention to edge cases.

Unlike regression tests, to which they are very similar, characterization tests do not verify the *correct* behaviour of the code, which can be impossible to determine. Instead they verify the behaviour that was observed when they were written. Often no specification or test suite is available, leaving only characterization tests as an option, since the conservative path is to assume that the old behaviour is the required behaviour. Characterization tests are, essentially, change detectors. It is up to the person analyzing the results to determine if the detected change was expected and/or desirable, or unexpected and/or undesirable.

One of the interesting aspects of characterization tests is that, since they are based on existing code, it's possible to generate some characterization tests automatically. An automated characterization test tool will exercise existing code with a wide range of relevant and/or random input values, record the output values (or state changes) and generate a set of characterization tests. When the generated tests are executed against a new version of the code, they will produce one or more failures/warnings if that version of the code has been modified in a way that changes a previously established behaviour.

References

[1] Feathers, Michael C. *Working Effectively with Legacy Code* (ISBN 0-13-117705-2).

External links

- Characterization Tests (<http://c2.com/cgi/wiki?CharacterizationTest>)
- Working Effectively With Characterization Tests (<http://www.artima.com/weblogs/viewpost.jsp?thread=198296>) first in a blog-based series of tutorials on characterization tests.
- Change Code Without Fear (<http://www.ddj.com/development-tools/206105233>) DDJ article on characterization tests.

Cloud testing

Cloud testing is a form of software testing in which web applications that use cloud computing environments (a "cloud") seek to simulate real-world user traffic as a means of load testing and stress testing web sites. The ability and cost to simulate web traffic for software testing purposes has been an inhibitor to overall web reliability. The low cost and accessibility of the cloud's extremely large computing resources provides the ability to replicate real world usage of these systems by geographically distributed users, executing wide varieties of user scenarios, at scales previously unattainable in traditional testing environments.

Companies simulate real world Web users by using cloud testing services that are provided by cloud service vendors such as SOASTA, HP, Load Impact, Compuware and Keynote systems. Once user scenarios are developed and the test is designed, these service providers leverage cloud servers (provided by cloud platform vendors such as Amazon.com, Google, Rackspace, etc.) to generate Web traffic that originates from around the world. Once the test is complete, the cloud service providers deliver results and analytics back to corporate IT professionals through real-time dashboards for a complete analysis of how their applications and networks will perform during peak volumes.

Testing in the cloud is often discussed in the context of performance or load tests against cloud-based applications. However, all types of software application tests, be they performance, functionality, usability, etc., can be regarded as cloud testing if the testing entity targets an application residing on a third-party computing platform, and accesses that platform via the Internet. Cloud computing itself is often referred to as the marriage of Software as a Service (SaaS) and Utility Computing. In regard to test execution, the software offered as a service may be a transaction generator and the cloud provider's infrastructure software, or may just be the latter.

Leading cloud computing service providers include, among others, Amazon, 3-terra, Skytap, HP and SOASTA. Some keys to successful testing in the cloud include

1. understanding a platform provider's elasticity model/dynamic configuration method,
2. staying abreast of the provider's evolving monitoring services and Service Level Agreements (SLAs),
3. potentially engaging the service provider as an on-going operations partner if producing commercial off-the-shelf (COTS) software, and
4. being willing to be used as a case study by the cloud service provider. The latter may lead to cost reductions.

An evolving cloud testing community is forming under the auspices of the Software Testing & Quality Assurance group hosted by LinkedIn. Testing professionals openly share their experiences and exchange ideas related to cloud testing in order to enhance one another's proficiencies.

External links

- [Computing in the Clouds](#) ^[1]
 - [Cloud computing shapes up as big trend for 2009](#) ^[2]
 - [Cloud testing White Paper](#) ^[3]
 - [HP LoadRunner in the Cloud](#) ^[4]
 - [CloudSleuth Testing Tools](#) ^[5]
-

References

- [1] <http://www.ddj.com/web-development/213000642>
- [2] http://www.infoworld.com/article/09/01/28/cloud-computing-shapes-up-as-big-trend-for-2009_1.html
- [3] http://www.cloud-intelligence.com/sites/www.cloud-intelligence.com/files/Cloud%20Testing%20White%20Paper_0.pdf
- [4] <http://www.hp.com/go/loadrunnercloud>
- [5] <https://www.cloudsleuth.net/web/guest/applications>

Code coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing.^[1]

Code coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in *Communications of the ACM* in 1963.^[2]

Code coverage is one consideration in the safety certification of avionics equipment. The standard by which avionics gear is certified by the Federal Aviation Administration (FAA) is documented in DO-178B.^[3]

Coverage criteria

To measure how well the program is exercised by a test suite, one or more *coverage criteria* are used.

Basic coverage criteria

There are a number of coverage criteria, the main ones being:^[4]

- **Function coverage** - Has each function (or subroutine) in the program been called?
- **Statement coverage** - Has each node in the program been executed?
- **Decision coverage** (not the same as **branch coverage**.^[5]) - Has every edge in the program been executed? For instance, have the requirements of each branch of each control structure (such as in IF and CASE statements) been met as well as not met?
- **Condition coverage** (or predicate coverage) - Has each boolean sub-expression evaluated both to true and false? This does not necessarily imply decision coverage.
- **Condition/decision coverage** - Both decision and condition coverage should be satisfied.

For example, consider the following C++ function:

```
int foo(int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

Assume this function is a part of some bigger program and this program was run with some test suite.

- If during this execution function 'foo' was called at least once, then *function coverage* for this function is satisfied.
- *Statement coverage* for this function will be satisfied if it was called e.g. as `foo(1, 1)`, as in this case, every line in the function is executed including `z = x;`.
- Tests calling `foo(1, 1)` and `foo(0, 1)` will satisfy *decision coverage*, as in the first case the `if` condition and the short circuit condition are satisfied and `z = x;` is executed, and in the second neither conditional is

satisfied and x is not assigned to z .

- *Condition coverage* can be satisfied with tests that call `foo(1, 1)`, `foo(1, 0)` and `foo(0, 0)`. These are necessary as in the first two cases ($x > 0$) evaluates to `true` while in the third it evaluates `false`. At the same time, the first case makes ($y > 0$) `true` while the second and third make it `false`.

In languages, like Pascal, where standard boolean operations are not short circuited, condition coverage does not necessarily imply decision coverage. For example, consider the following fragment of code:

```
if a and b then
```

Condition coverage can be satisfied by two tests:

- `a=true, b=false`
- `a=false, b=true`

However, this set of tests does not satisfy decision coverage as in neither case will the `if` condition be met.

Fault injection may be necessary to ensure that all conditions and branches of exception handling code have adequate coverage during testing.

Modified condition/decision coverage

For safety-critical applications (e.g., for avionics software) it is often required that **modified condition/decision coverage (MC/DC)** is satisfied. This criteria extends condition/decision criteria with requirements that each condition should affect the decision outcome independently. For example, consider the following code:

```
if (a or b) and c then
```

The condition/decision criteria will be satisfied by the following set of tests:

- `a=true, b=true, c=true`
- `a=false, b=false, c=false`

However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output. So, the following test set is needed to satisfy MC/DC:

- **a=false, b=false, c=true**
- **a=true, b=false, c=true**
- **a=false, b=true, c=true**
- **a=true, b=true, c=false**

The bold values influence the output, each variable must be present as an influencing value at least once with false and once with true.

Multiple condition coverage

This criteria requires that all combinations of conditions inside each decision are tested. For example, the code fragment from the previous section will require eight tests:

- `a=false, b=false, c=false`
- `a=false, b=false, c=true`
- `a=false, b=true, c=false`
- `a=false, b=true, c=true`
- `a=true, b=false, c=false`
- `a=true, b=false, c=true`
- `a=true, b=true, c=false`
- `a=true, b=true, c=true`

Other coverage criteria

There are further coverage criteria, which are used less often:

- **Linear Code Sequence and Jump (LCSAJ) coverage** - has every LCSAJ been executed?
- **JJ-Path coverage** - have all jump to jump paths ^[6] (aka LCSAJs) been executed?
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?
- **Loop coverage** - Has every possible loop been executed zero times, once, and more than once?

Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage.

Some of the coverage criteria above are connected. For instance, path coverage implies decision, statement and entry/exit coverage. Decision coverage implies statement coverage, because every statement is part of a branch.

Full path coverage, of the type described above, is usually impractical or impossible. Any module with a succession of n decisions in it can have up to 2^n paths within it; loop constructs can result in an infinite number of paths. Many paths may also be infeasible, in that there is no input to the program under test that can cause that particular path to be executed. However, a general-purpose algorithm for identifying infeasible paths has been proven to be impossible (such an algorithm could be used to solve the halting problem).^[7] Methods for practical path coverage testing instead attempt to identify classes of code paths that differ only in the number of loop executions, and to achieve "basis path" coverage the tester must cover all the path classes.

In practice

The target software is built with special options or libraries and/or run under a special environment such that every function that is exercised (executed) in the program(s) is mapped back to the function points in the source code. This process allows developers and quality assurance personnel to look for parts of a system that are rarely or never accessed under normal conditions (error handling and the like) and helps reassure test engineers that the most important conditions (function points) have been tested. The resulting output is then analyzed to see what areas of code have not been exercised and the tests are updated to include these areas as necessary. Combined with other code coverage methods, the aim is to develop a rigorous, yet manageable, set of regression tests.

In implementing code coverage policies within a software development environment one must consider the following:

- What are coverage requirements for the end product certification and if so what level of code coverage is required? The typical level of rigor progression is as follows: Statement, Branch/Decision, Modified Condition/Decision Coverage(MC/DC), LCSAJ (Linear Code Sequence and Jump)
- Will code coverage be measured against tests that verify requirements levied on the system under test (DO-178B)?
- Is the object code generated directly traceable to source code statements? Certain certifications, (i.e. DO-178B Level A) require coverage at the assembly level if this is not the case: "Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences" (DO-178B) para-6.4.4.2.^[3]

Test engineers can look at code coverage test results to help them devise test cases and input or configuration sets that will increase the code coverage over vital functions. Two common forms of code coverage used by testers are statement (or line) coverage and path (or edge) coverage. Line coverage reports on the execution footprint of testing in terms of which lines of code were executed to complete the test. Edge coverage reports which branches or code decision points were executed to complete the test. They both report a coverage metric, measured as a percentage. The meaning of this depends on what form(s) of code coverage have been used, as 67% path coverage is more comprehensive than 67% statement coverage.

Generally, code coverage tools and libraries exact a performance and/or memory or other resource cost which is unacceptable to normal operations of the software. Thus, they are only used in the lab. As one might expect, there are classes of software that cannot be feasibly subjected to these coverage tests, though a degree of coverage mapping can be approximated through analysis rather than direct testing.

There are also some sorts of defects which are affected by such tools. In particular, some race conditions or similar real time sensitive operations can be masked when run under code coverage environments; and conversely, some of these defects may become easier to find as a result of the additional overhead of the testing code.

Software tools

Tools for C / C++

- Cantata++
- DevPartner
- Gcov ^[8] with graphical summaries LCOV ^[9] and text/XML summaries gcovr ^[10]
- Insure++
- NuMega TrueCoverage
- LDRA Testbed
- Tessy
- Testwell CTC++
- Trucov

Tools for C# .NET

- DevPartner
- JetBrains dotCover ^[11]
- Kalistick
- NCover
- TestDriven.NET ^[12]
- Visual Studio 2010 ^[13]

Tools for Java

- Cobertura ^[14]
 - Clover
 - DevPartner
 - EMMA
 - Jtest
 - Kalistick
 - LDRA Testbed
 - Serenity
-

Tools for Perl

- Devel::Cover^[15] is a complete suite for generating code coverage reports in HTML and other formats.

Tools for PHP

- PHPUnit, also need Xdebug to make coverage reports

Tools for Python

- Coverage.py^[16]
- Figleaf^[17]

Hardware tools

- Aldec
- Atrenta
- Cadence Design Systems
- JEDA Technologies
- Mentor Graphics
- Nusym Technology
- Simucad Design Automation
- Synopsys

References

- [1] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. p. 254. ISBN 0470042125. .
- [2] Joan C. Miller, Clifford J. Maloney (February 1963). "Systematic mistake analysis of digital computer programs" (<http://doi.acm.org/10.1145/366246.366248>). *Communications of the ACM* (New York, NY, USA: ACM) **6** (2): 58–63. doi:10.1145/366246.366248. ISSN 0001-0782. .
- [3] RTCA/DO-178B, *Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics, December 1, 1992.
- [4] Glenford J. Myers (2004). *The Art of Software Testing, 2nd edition*. Wiley. ISBN 0471469122.
- [5] Position Paper CAST-10 (June 2002). *What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)?* (http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf)
- [6] M. R. Woodward, M. A. Hennell, "On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC", *Information and Software Technology* 48 (2006) pp. 433-440
- [7] Dorf, Richard C.: *Computers, Software Engineering, and Digital Devices*, Chapter 12, pg. 15. CRC Press, 2006. ISBN 0849373409, 9780849373404; via Google Book Search ([http://books.google.com/books?id=jykvITCoksMC&pg=PT386&lpg=PT386&dq="infeasible+path"+"halting+problem"&source=web&ots=WUWz3qMPRv&sig=dSAjrlHBSZJcKWZfGa_IxYlfSNA&hl=en&sa=X&oi=book_result&resnum=1&ct=result](http://books.google.com/books?id=jykvITCoksMC&pg=PT386&lpg=PT386&dq=))
- [8] <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html>
- [9] <http://ltp.sourceforge.net/coverage/lcov.php>
- [10] <https://software.sandia.gov/trac/fast/wiki/gcovr>
- [11] <http://www.jetbrains.com/dotcover/>
- [12] <http://testdriven.net/default.aspx>
- [13] <http://msdn.microsoft.com/en-us/library/ms182496.aspx>
- [14] <http://cobertura.sourceforge.net/>
- [15] <http://search.cpan.org/perldoc?Devel::Cover>
- [16] <http://nedbatchelder.com/code/coverage/>
- [17] <http://darcs.idyll.org/~t/projects/figleaf/doc/>

External links

- Branch Coverage for Arbitrary Languages Made Easy (<http://www.semdesigns.com/Company/Publications/TestCoverage.pdf>)
- Code Coverage Analysis (<http://www.bullseye.com/coverage.html>) by Steve Cornett
- Code Coverage Introduction (<http://www.javaranch.com/newsletter/200401/IntroToCodeCoverage.html>)
- Development Tools (Java)/ Code coverage (http://www.dmoz.org//Computers/Programming/Languages/Java/Development_Tools/Performance_and_Testing/Code_Coverage) at the Open Directory Project
- Development Tools (General)/ Code coverage (http://www.dmoz.org//Computers/Programming/Software_Testing/Products_and_Tools) at the Open Directory Project
- FAA CAST Position Papers (http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/)

Code integrity

Code integrity is a measurement used in software testing. It measures the how high is the source code's quality when it is passed on to the QA, and is affected by how extensively the code was unit tested and integration tested. Code integrity is a combination of code coverage and software quality, and is usually achieved by unit testing your code to reach high code coverage.

With code integrity, the developer can be sure that his code is written correctly when passed on to QA. This is, in fact, the expected quality level of the code. Code integrity helps companies release better products, with fewer bugs, in a shorter time.

Companies who practice code integrity avoid the classic scenario where the development stage is delayed, delaying the QA stage, delaying the release stage. The product is released with more bugs (due to time pressure), users report tons of bugs back to the development team, and they start working on version 1.1 shortly after releasing version 1.0, just to fix bugs that could have been avoided.

The QA department can't measure the code's integrity even after all their tests are run. The only way to measure code integrity, and be sure of your code, is by unit testing your code, and reaching high code coverage.

Improve Code integrity by:

- Unit testing the code
- Integration testing
- Assigning a code integrity manager

Advantages of working with code integrity:

- Shorter development time - bugs that are found during the development stage are fixed faster and easier than bugs found in later stages.
- Lower development costs – It's cheaper to fix bugs that are found during the development stage than in later stages.
- Confidence in your code's quality – Releasing products with high code integrity means more positive feedback from your customers.
- Makes the QA's work much more efficient – The QA concentrates on testing the system, without worrying about bugs that could have been easily found through proper unit testing.

Measuring code integrity:

To measuring code integrity, use the following formula: $1 - (\text{Non-covered bugs})/(\text{Total bugs})$


In words:, the 100% code integrity minus the number of bugs that weren't covered by unit testing, divided by the total bugs found during the entire product cycle., including development is the code not in integrity.

Codenomicon

Codenomicon

Type	Privately held company
Founded	2001
Headquarters	Oulu, Finland
Area served	worldwide
Products	Robustness Testing Tools, Situation Awareness Tools
Services	Security Testing Services, Network Analysis Services
Owner(s)	Private (profitable since 2008)
Employees	70
Website	http://www.codenomicon.com/

Codenomicon

	
Operating system	Cross-platform
Type	Computer security, Fuzzing, Robustness testing, Network Analysis
Website	http://www.codenomicon.com/

Codenomicon is a private company founded in late 2001, and develops robustness testing tools (also called fuzzing tools) for manufacturers, service providers, government/defense and enterprise customers. The company has raised Venture money mid 2000's and has been profitable since 2008, with more than 40% growth in sales each year.^[1] In 2011, the company acquired Clarified Networks, a situation awareness company.^[2]

Codenomicon is based in Oulu, Finland (Europe), and has offices in Saratoga, California (US), Hong Kong (Asia/Pacific) and Singapore (Asia/Pacific).^[3]

Codenomicon is also known for having t-shirts that say "GO HACK YOURSELF", which they usually have at their booth during security conferences. This comes from the goal of Codenomicon to enable testers and system administrators to find their own zero-day vulnerabilities, instead of depending on external security consultants, and special hacker skills.

Products

The product line of Codenomicon consists of a suite of 200+ independent network protocol testing solutions called DEFENSICS. Each protocol fuzzer can be licensed separately, or as a suite of protocols related to a specific technology such as IPTV, VoIP, Routing, Bluetooth, and several other communication domains.^[4]

These tools have roots in the research done at the University of Oulu in the Secure Programming Group (OUSPG).^[5] Whereas since 1999 the PROTOS project produced free software for testing about 10 protocols, Codenomicon has added support for much wider test coverage for about 200+ protocols, and is providing those tools under commercial licensing. PROTOS tools are still widely used.^[6] PROTOS and Codenomicon testing approach, called robustness testing, is based around the idea of proactive protocol testing by injecting unexpected anomalies into the protocol message sequences, structures and data types; in essence, fuzzing with some intelligence behind the generated test data.

DEFENSICS includes test suites for 200+ protocols industry standard networks protocols such as SMTP, SNMP, BGP, IPv6, SSH and SIP. In addition there are also test suites for various Bluetooth profiles and Wireless LAN.^[7] Codenomicon has also built nearly 100 customer proprietary fuzzers for special interfaces such as device API's and complex banking systems.

Robustness testing

Robustness testing is a model based fuzzing technique and over all Black box testing, an extension of syntax testing, that systematically will explore the input space defined by various communication interfaces or data formats, and will generate intelligent test cases that find crash-level flaws and other failures in software.^[8] The technique was first described in a University of Oulu white paper on robustness testing published in 2000, by Kaksonen et al.,^[9] and Licentiate Thesis by Kaksonen,^[10] published in 2001. Fault injection and specification mutations were other names they used for the same approach.^[11]

Codemicon's Defensics Product line is also known as a "Fuzzer that does not fuzz"^[12] - means - it uses smart anomalies instead of random Fuzzing structures. This enables fast test execution, extensive test documentation and better test coverage. Defensics tools address all fields in the protocols with all effective combinations of anomalies. Traditional fuzzing lacks this capability as with random inputs that would take too much time to be effective in fast paced test cycles.

History

Codemicon and its founders have been developing fuzzing tools since 1996.

The first ideas for the engine were based on ideas the founders had while working at OUSPG, where systematic fuzzing was first used to break ASCII/MIME contents in email clients and web services.^[13] ^[14] Later, the same technique was applied to ASN.1 structures in such protocols as SNMP, LDAP and X.509.^[15] ^[16]

After Codenomicon was founded in 2001, its DEFENSICS product line has grown to cover over 200 industry-standard network protocols and file formats, including wireless interfaces such as Bluetooth and WLAN. DEFENSICS for XML provides an added capability for testing common XML-based protocols and file formats more efficiently than before.^[17]

After founding Codenomicon, also PROTOS Test-Suites disclose they are running on top of Codenomicon engine.^[18] The research side span out into PROTOS Genome.^[19]

References

- [1] Codonomicon Newsletter 2010/12 (<http://www.codonomicon.com/news/newsletter/archive/2010-12.html#1>)
- [2] Acquisition Expands Codonomicon's Offering of Proactive Defense Solutions. News on EON. (<http://eon.businesswire.com/news/eon/20110523005695/en>)
- [3] Codonomicon history (<http://www.codonomicon.com/company/history.shtml>)
- [4] <http://www.codonomicon.com/products/test-suites.shtml> Codonomicon Test Suite Catalogue
- [5] OUSPG (<http://www.ee.oulu.fi/research/ouspg>)
- [6] PROTOS (<http://www.ee.oulu.fi/research/ouspg/protos>)
- [7] Codonomicon DEFENSICS Test Suites (<http://www.codonomicon.com/products/test-suites.shtml>)
- [8] LWN Security (<http://lwn.net/Articles/228366/>)
- [9] Kaksonen R., Laakso M., Takanen A. Vulnerability Analysis of Software through Syntax Testing. White paper. OUSPG 2001. (https://www.ee.oulu.fi/research/ouspg/PROTOS_WP2000-robustness)
- [10] Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Published in 2001 by Technical Research Centre of Finland, VTT Publications 447. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.). (<http://www.vtt.fi/inf/pdf/publications/2001/P448.pdf>)
- [11] Kaksonen R., Laakso M., Takanen A.. "Software Security Assessment through Specification Mutations and Fault Injection". In Proceedings of Communications and Multimedia Security Issues of the New Century / IFIP TC6/TC11 Fifth Joint Working Conference on Communications and Multimedia Security (CMS'01) May 21-22, 2001, Darmstadt, Germany; edited by Ralf Steinmetz, Jana Dittmann, Martin Steinebach. ISDN 0-7923-7365-0. (<http://www.ee.oulu.fi/research/ouspg/protos/analysis/CMS2001-spec-centered/>)
- [12] <http://crashatime.blogspot.com/2009/08/fuzzer-that-does-not-fuzz.html> The Fuzzer That Does Not Fuzz
- [13] Mime bugs in Netscape. (<http://netscape.1command.com/relnotes/details.phtml#10>)
- [14] The buzz on the bug - How does the e-mail security bug affect Solaris users? By Stephanie Steenbergen, SunWorld staff. (<http://sunsite.uakom.sk/sunworldonline/swol-08-1998/swol-08-emailbug.html>)
- [15] CERT Advisory CA-2001-18 Multiple Vulnerabilities in Several Implementations of the Lightweight Directory Access Protocol (LDAP). (<http://www.cert.org/advisories/CA-2001-18.html>)
- [16] Edmund Whelan. SNMP and Potential ASN.1 Vulnerabilities. December 2002. SANS Institute InfoSec Reading Room. (http://www.sans.org/reading_room/whitepapers/protocols/snmp_and_potential_asn_1_vulnerabilities_912)
- [17] XML Security and Fuzzing. <http://www.codonomicon.com/labs/xml/>
- [18] Bryan Burns, Jennifer Granick, Steve Manzuik, Dave Killion, Paul Guersch, Nicolas Beauchesne. Security Power Tools. Published by O'Reilly. ([http://books.google.com/books?q="Created with Codonomicon Mini-Simulation Toolkit"](http://books.google.com/books?q=))
- [19] Viide J., Helin A., Laakso M., Pietikäinen P., Seppänen M., Halunen K., Puuperä R., Röning J. "Experiences with Model Inference Assisted Fuzzing". In proceedings of the 2nd USENIX Workshop on Offensive Technologies (WOOT '08). San Jose, CA. July 28, 2008. (<http://www.ee.oulu.fi/research/ouspg/protos/sota/woot08-experiences/>)

External links

- Official site (<http://www.codonomicon.com/>)
- AlwaysOn as an 100 Top Private Company Award Winner (<http://www.prweb.com/releases/2007/8/prweb545287.htm>)
- eSecurity DEFEND THEN DEPLOY. (<http://www.techguideonline.com/esecurity/datasheet.php?ds=21>)
- Codonomicon Introduces DEFENSICS for WLAN (<http://www.tmcnet.com/wifirevolution/articles/13638-codonomicon-introduces-defensics-wlan.htm>)
- Codonomicon Offers Preemptive Security and Quality Testing (<http://www.itcinstitute.com/display.aspx?id=4632>)
- CODENOMICON DEFENDS AGAINST NETWORK DATA STORAGE THREATS (http://www.bapcojournal.com/news/fullstory.php/aid/361/CODENOMICON_DEFENDS_AGAINST_NETWORK_DATA_STORAGE_THREATS.html)
- Jolt Productivity Award 2008 (<http://www.joltawards.com/press/030608.htm>)
- Dr. Dobbs interview with Ari Takanen: Fuzzing, Model-based Testing, and Security <http://www.drdoobbs.com/security/207000941>
- Dr. Dobbs article on Automated Penetration Testing Toolkit Released (based on Codenomicon press release) <http://www.drdoobbs.com/security/224600546>

Security advisory links

- Codenomicon Advisories (<http://www.codenomicon.com/labs/advisories/>)
- CERT-FI Advisory on XML libraries (<https://www.cert.fi/en/reports/2009/vulnerability2009085.html>)
- CERT-FI Vulnerability Advisory on GnuTLS (<http://www.cert.fi/haavoittuvuudet/advisory-gnutls.html>)
- CVE-2004-0786 (<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0786>)
- CVE-2004-0081 (<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0081>)

Video links

- Heikki Kortti - Designing Inputs That Make Software Fail (<http://video.google.com/videoplay?docid=6509883355867972121>)
- Codenomicon - HS Startup competition Video (<http://www.viddler.com/explore/antti/videos/9/>)

Compatibility testing

Compatibility testing, part of software non-functional tests, is testing conducted on the application to evaluate the application's compatibility with the computing environment. Computing environment may contain some or all of the below mentioned elements:

- Computing capacity of Hardware Platform (IBM 360, HP 9000, etc.)..
- Bandwidth handling capacity of networking hardware
- Compatibility of peripherals (Printer, DVD drive, etc.)
- Operating systems (MVS, UNIX, Windows, etc.)
- Database (Oracle, Sybase, DB2, etc.)
- Other System Software (Web server, networking/ messaging tool, etc.)
- Browser compatibility (Firefox, Netscape, Internet Explorer, Safari, etc.)

Browser compatibility testing, can be more appropriately referred to as user experience testing. This requires that the web applications are tested on different web browsers, to ensure the following:

- Users have the same visual experience irrespective of the browsers through which they view the web application.
- In terms of functionality, the application must behave and respond the same way across different browsers.

For more information please visit the link BCT ^[1]

- Carrier compatibility (Verizon, Sprint, Orange, O2, AirTel, etc.)
- Backwards compatibility.
- Hardware (different phones)
- Different Compilers (compile the code correctly)
- Runs on multiple host/guest Emulators

Certification testing falls within the scope of Compatibility testing. Product Vendors run the complete suite of testing on the newer computing environment to get their application certified for a specific Operating Systems or Databases.

References

[1] <http://essentiaserve.com/bct.html>

Component-Based Usability Testing

Component-based usability testing (CBUT) is a testing approach which aims at empirically testing the usability of an interaction component. The latter is defined as an elementary unit of an interactive system, on which behaviour-based evaluation is possible. For this, a component needs to have an independent, and by the user perceivable and controllable state, such as a radio button, a slider or a whole word processor application. The CBUT approach can be regarded as part of component-based software engineering branch of software engineering.

Theory

CBUT is based on both software architectural views such as Model–View–Controller (MVC), Presentation-Abstraction-Control (PAC), ICON and CNUCE agent models that split up the software in parts, and cognitive psychology views where a person's mental process is split up in smaller mental processes. Both software architecture and cognitive architecture use the principle of hierarchical layering, in which low level processes are more elementary and for humans often more physical in nature, such as the coordination movement of muscle groups. Processes that operate on higher level layers are more abstract and focus on a person's main goal, such as writing an application letter to get a job. The Layered Protocol Theory^[1] (LPT), which is a special version of Perceptual Control Theory (PCT), brings these views together by suggesting that users interact with a system across several layers by sending messages. Users interact with components on high layers by sending messages, such as pressing keys, to components operating on lower layers, which on their turn relay a series of these messages into a single high level message, such as 'DELETE *.*', to a component on a higher layer. Components operating on higher layers, communicate back to the user by sending messages to components operating on lower level layers. Whereas this layered-interaction model explains how the interaction is established, control loops explain the purpose of the interaction. LPT sees the purpose of the users' behaviour as the users' attempt to control their perception, in this case the state of the component they perceive. This means that users will only act if they perceive the component to be in an undesirable state. For example, if a person have an empty glass but want a full glass of water, he or she will act (e.g. walk to the tap, turning the tap on to fill the glass). The action of filling the glass will continue until the person perceives the glass as full. As interaction with components takes places on several layers, interacting with a single device can include several control loops. The amount of effort put into operating a control loop is seen as an indicator for the usability of an interaction component.

Testing

CBUT can be categorized according to two testing paradigms, the Single-Version Testing Paradigm (SVTP) and the Multiple-Versions Testing Paradigm (MVTP). In SVTP only one version of each interaction component in a system is tested. The focus is to identify interaction components that might reduce the overall usability of the system. SVTP is therefore suitable as part of a software-integration test. In MVTP on the other hand, multiple versions of a single component are tested while the remaining components in the system remain unchanged. The focus is on identifying the version with the highest usability of specific interaction component. MVTP therefore is suitable for component development and selection. Different CBUT methods have been proposed for SVTP and MVTP, which include measures based on recorded user interaction and questionnaires. Whereas in MVTP the recorded data can directly be interpreted by making a comparison between two versions of the interaction component, in SVTP log file analysis is more extensive as interaction with both higher and lower components must be considered^[2]. Meta-analysis on the data from several lab experiments that used CBUT measures suggests that these measures can be statistically more powerful than overall (holistic) usability measures^[3].

Usability questionnaire

Whereas holistic oriented usability questionnaires such as System Usability Scale (SUS) examine the usability of a system on several dimensions such as defined in ISO 9241 Part 11 standard effectiveness, efficiency and satisfaction, a Component-Based Usability Questionnaire (CBUQ) ^[4] is a questionnaire which can be used to evaluate the usability of individual interaction components, such as the volume control or the play control of a MP3 player. To evaluate an interaction component, the six Perceived Ease-Of-Use (PEOU) statements from the Technology acceptance model are taken with a reference to the interaction component, instead of to the entire system, for example:

Learning to operate the Volume Control would be easy for me.

Users are asked to rate these statements on a seven point Likert Scale. The average rating on these six statements is regarded as the user's usability rating of the interaction component. Based on lab studies with difficult to use interaction components and easy to use interaction components, a break-even point of 5.29 on seven point Likert scale has been determined^[4]. Using a One-sample student's t-test it is possible to examine whether users' rating of an interaction component deviates from this break-even point. Interaction components that receive rating below this break-even point can be regarded as more comparable to the set of difficult to use interaction components, whereas ratings above this break-even point would be more comparable to the set if easy to use interaction components.

If engineers like to evaluate multiple interaction components simultaneously, the CBUQ questionnaire exists of separate sections, one for each interaction component, each with their own 6 PEOU statements.

References

- [1] Farrell, P.S.E., Hollands, J.G., Taylor, M.M., Gamble, H.D., (1999). Perceptual control and layered protocols in interface design: I. Fundamental concepts. *International Journal of Human-Computer Studies* 50 (6), 489-520. online (<http://dx.doi.org/doi:10.1006/ijhc.1998.0259>)
- [2] Brinkman, W.-P., Haakma, R., & Bouwhuis, D.G. (2007), Towards an empirical method of efficiency testing of system parts: a methodological study, *Interacting with Computers*, vol. 19, no. 3, pp. 342-356. preliminary version (http://mmi.tudelft.nl/~willem-paul/WP_Papers_online_versie/Towards_an_empirical_method_of_efficiency_testing_of_system_parts_a_methodological_study_preliminary_version.pdf) online (<http://dx.doi.org/doi:10.1016/j.intcom.2007.01.002>)
- [3] Brinkman, W.-P., Haakma, R., & Bouwhuis, D.G. (2008). Component-Specific Usability Testing, *IEEE Transactions on Systems, Man, and Cybernetics - Part A*, vol. 38, no. 5, pp. 1143-1155, September 2008. preliminary version (http://mmi.tudelft.nl/~willem-paul/WP_Papers_online_versie/Component_specific_usability_testing_preliminary_version.pdf) online (<http://dx.doi.org/doi:10.1109/TSMCA.2008.2001056>)
- [4] Brinkman, W.-P., Haakma, R., & Bouwhuis, D.G. (2009), Theoretical foundation and validity of a component-based usability questionnaire, *Behaviour and Information Technology*, 2, no. 28, pp. 121 - 137. preliminary version (http://mmi.tudelft.nl/~willem-paul/WP_Papers_online_versie/The_theoretical_foundation_and_validity_of_a_component_based_usability_questionnaire_preliminary_version.pdf) MP3 example study (<http://mmi.tudelft.nl/~willem-paul/mp3player/Intro.htm>) online (<http://dx.doi.org/DOI:10.1080/01449290701306510>)

External links

- Example study (<http://mmi.tudelft.nl/~willem-paul/mp3player/Intro.htm>) that uses Component-based Usability Questionnaire including instructions, questionnaires (<http://mmi.tudelft.nl/~willem-paul/mp3player/study.htm>), data analysis (<http://mmi.tudelft.nl/~willem-paul/mp3player/results.htm>) and additional instructions (<http://mmi.tudelft.nl/willem-paul/index.php/Questionnaires>) .

Conference Room Pilot

Conference Room Pilot (CRP) is a term used in software procurement and software acceptance testing. A CRP may be used during the selection and implementation of a software application in an organisation or company.

The purpose of the Conference Room Pilot is to validate a software application against the business processes of end-users of the software, by allowing end-users to use the software to carry out typical or key business processes using the new software. A commercial advantage of a Conference Room Pilot is that it may allow the customer to prove that the new software will do the job (meets business requirements and expectations) before committing to buying the software, thus avoiding buying an inappropriate application. The term is most commonly used in the context of 'out of the box' (OOTB) or 'commercial off-the-shelf' software (COTS).

Compared to User Acceptance Testing

Although a Conference Room Pilot shares some features of User Acceptance Testing (UAT), it should not be considered a testing process - it validates Design or Solution fit for purpose at a higher level than functional testing.

Shared features of CRP and UAT include:

- End-to-end business processes are used as a "business input" for both
- Functionality demonstrations
- Non-functional validation(e.g. performance testing)

Differences between a Conference Room Pilot and a formal UAT:

- It is attempting to identify how well the application meets business needs, and identify gaps, whilst still in the design phase of the project
- There is an expectation that changes will be required before acceptance of the solution
- The software is 'on trial' and may be rejected completely in favour of another solution.

References

- http://www.ensync-corp.com/consulting/conference_room_pilot.cfm?section=consulting
 - http://www-archive.ui-integrate.uillinois.edu/news_art_crp.asp
 - <http://www.bourkeconsulting.com/documents/POCCRPBCAWebsite020903.pdf>
 - http://www.smthacker.co.uk/conference_room_pilot.htm
-

Conformance testing

Conformance testing or **type testing** is testing to determine whether a product or system meets some specified standard that has been developed for efficiency or interoperability.

To aid in this, many test procedures and test setups have been developed, either by the standard's maintainers or external organizations, specifically for testing conformance to standards.

Conformance testing is often performed by external organizations, which is sometimes the standards body itself, to give greater guarantees of compliance. Products tested in such a manner are then advertised as being certified by that external organization as complying with the standard.

Service providers, equipment manufacturers, and equipment suppliers rely on this data to ensure Quality of Service (QoS) through this conformance process.

Software engineering

In software testing, Compilers, for instance, are extensively tested to determine whether they meet the recognized standard for that language. It is a process of testing an implemented product to confirm that it is based on its specified standards.

Electronic and electrical engineering

In electronic engineering and electrical engineering, some countries and business environments (such as telecommunication companies) require that an electronic product meet certain requirements before they can be sold. Standards for telecommunication products written by standards organizations such as ANSI, the FCC, and IEC, etc., have certain criteria that a product must meet before compliance is recognized. In countries such as Japan, China, Korea, and some parts of Europe, products cannot be sold unless they are known to meet those requirements specified in the standards. Usually, manufacturers set their own requirements to ensure product quality, sometimes with levels much higher than what the governing bodies require. Compliance is realized after a product passes a series of tests without occurring some specified mode of failure. Failure levels are usually set depending on what environment the product will be sold in. For instance, test on a product for used in an industrial environment will not be as stringent as a product used in a residential area. A failure can include data corruption, loss of communication, and irregular behavior.

There are three main types of compliance test for electronic devices, emissions tests, immunity tests, and safety tests. Emissions tests ensure that a product will not emit harmful interference by electromagnetic radiation and/or electrical signals in communication and power lines. Immunity tests ensure that a product is immune to common electrical signals and Electromagnetic interference (EMI) that will be found in its operating environment, such as electromagnetic radiation from a local radio station or interference from nearby products. Safety tests ensure that a product will not create a safety risk from situations such as a failed or shorted power supply, blocked cooling vent, and powerline voltage spikes and dips.

Common Tests - refer to GR-1089, *Electromagnetic Compatibility and Electrical Safety - Generic Criteria for Network Telecommunications Equipment*

Radiated Immunity - refer to GR-1089, *Electromagnetic Compatibility and Electrical Safety - Generic Criteria for Network Telecommunications Equipment*, Section 3.3

An antenna is used to subject the device to electromagnetic waves, covering a large frequency range (usually from 30 MHz to 2.9 GHz).

Radiated Emissions

One or more antennas are used to measure the amplitude of the electromagnetic waves that a device emits. The amplitude must be under a set limit, with the limit depending on the devices classification.

Conducted Immunity

Low frequency signals (usually 10 kHz to 80 MHz) are injected onto the data and power lines of a device. This test is used to simulate the coupling of low frequency signals onto the power and data lines, such as from a local AM radio station.

Conducted Emissions

Similar to radiated emissions, except the signals are measured at the power lines with a filter device.

Electrostatic discharge (ESD) Immunity

Electrostatic discharges with various properties (rise time, peak voltage, fall time, and half time) are applied to the areas on the device that are likely to be discharged too, such as the faces, near user accessible buttons, etc. Discharges are also applied to a vertical and horizontal ground plane to simulate an ESD event on a nearby surface. Voltages are usually from 2kV to 15kV, but commonly go as high as 25kV or more.

Burst Immunity

Bursts of high voltage pulses are applied to the powerlines to simulate events such as repeating voltage spikes from a motor.

Powerline Dip Immunity

The line voltage is slowly dropped down then brought back up.

Powerline Surge Immunity


A surge is applied to the line voltage.

Telecom and datacom protocols

In protocol testing, TTCN-3 has been used successfully to deploy a number of test systems, including protocol conformance testers for SIP, WiMAX, and DSRC.

Core Security

Core Security

	
Type	Private
Industry	Computer Security Vulnerability Management Security Consulting Services
Founded	1996
Headquarters	Boston, MA and Buenos Aires, Argentina
Key people	Mark Hatton (President and CEO)
Employees	~ 200
Website	www.coresecurity.com ^[1]

Core Security Technologies is a computer and network security company that provides IT security testing and measurement software products and services. The company's research arm, CoreLabs, proactively identifies new IT security vulnerabilities, publishes public vulnerability advisories, and works with vendors to assist in eliminating the exposures they find.^[2]

History

- 1996: Core Security was founded in Buenos Aires, Argentina.
- 1997: The CoreLabs Research group was established and published their first advisory.
- 1998: Core conducted its first penetration test for a U.S. company.
- 1998: Core Security was recognized as an "Endeavor Entrepreneur" by the Endeavor Foundation, a foundation that supports entrepreneurial projects in emerging markets.
- 2000: The company's first U.S. office opened in New York, NY.
- 2002: Core released the first and second versions of their flagship penetration testing product, Core Impact Pro.^[3]
- 2003: The company's U.S. headquarters was relocated from New York to Boston, MA.
- 2008: Mark Hatton becomes CEO of Core Security.^[4]
- 2009: Core adds development sites in Boston and India.
- 2010: Core announces beta of new security testing and measurement product, Core Insight Enterprise

Company

Management Team^[5]

- Mark Hatton- President and Chief Executive Officer^[6]
- John O'Brien- Executive Vice President of Corporate Operations and CFO
- Ivan Arce- Chief Technology Officer
- Milan Shah- Senior Vice President of Engineering
- Jeffrey Cassidy- Vice President and General Manager of South American Operations
- Tom Kellermann- VP of Security Awareness and Government Affairs
- Kimberly Legelis- Vice President of Marketing
- Stephen Pace- Vice President of Sales and Services
- Fred Pinkett- Vice President of Product Management
- Paula Varas- Vice President of Engineering
- Ariel Waissbein- Director of Research & Development
- Alberto Soliño- Director of Security Consulting Services

Board of Directors^[7]

- Jonatan Altszul- Co-Founder, Core Security Technologies and Managing Director of Aconcagua Ventures
- Shinya Akamine- CEO, BlueRoads Corp.
- Jeronimo Bosch- Pegasus Capital
- Peter Chung- Morgan Stanley Venture Partners
- Edward Hamburg- Morgan Stanley Venture Partners
- Mark Hatton- President and CEO, Core Security Technologies
- Robert Steinkrauss- CEO, ChosenSecurity, Inc

Advisory Board

The Core Advisory Board helps to guide the company's business strategy, vulnerability research and product development plans.^[8]

- Roland Cloutier, Vice President, Chief Security Officer, ADP Corp.
- Melissa Hathaway, President of Hathaway Global Strategies, LLC and Former Acting Senior Director for Cyberspace for the National Security and Homeland Security Councils^[9]
- John Stewart, Vice President and Chief Security Office, Cisco^[10]

Products

Core Impact Pro: a penetration testing software solution that replicates cyber attacks to assess the security of web applications, network systems, endpoint systems, email users and wireless networks^{[11] [12]}

Core Insight Enterprise: a security testing and measurement product for large environments that will be available in late 2010^[13]

Services

Security Consulting Services: in-depth penetration testing and source code auditing services

CORE IMPACT Professional Services: penetration testing services based on the company's CORE IMPACT product^[14]

CORE IMPACT Certified Professional (CICP) Training and Certification: advanced penetration testing training for IT security practitioners and consultants^[15]

Facilities

Core Security is headquartered in Boston, MA, and Buenos Aires, Argentina.

Research and Advisories

Core Security's research department, CoreLabs, conducts research in various areas of computer security, including system vulnerabilities, cyber attack planning and simulation, source code auditing and cryptography. CoreLabs regularly publishes security advisories, technical papers, project information and shared software tools for public use, with its researchers participating in many IT security research conferences including the Black Hat Briefings [16] [17]

Recent Advisories

- Microsoft Windows CreateWindow function callback vulnerability^[18]
- Microsoft Office Excel PivotTable Cache Data Record Buffer Overflow^[19]
- HP OpenView NNM OvJavaLocale Buffer Overflow Vulnerability^[20]

References

- [1] <http://www.coresecurity.com>
- [2] <http://www.bizjournals.com/boston/stories/2010/08/30/story11.html>
- [3] <http://www.coresecurity.com/content/core-security-technologies-announces-first-comprehensive-penetration-testing>
- [4] <http://www.coresecurity.com/content/core-security-technologies-appoints-new-ceo-relocates-corporate>
- [5] <http://www.coresecurity.com/content/management-team>
- [6] <http://www.coresecurity.com/content/core-security-technologies-appoints-new-ceo-relocates-corporate>
- [7] <http://www.coresecurity.com/content/board-of-directors>
- [8] <http://www.coresecurity.com/content/core-advisory-board>
- [9] http://newsroom.cisco.com/dlls/2010/prod_020210c.html
- [10] <http://www.coresecurity.com/content/board-of-directors>
- [11] <http://www.coresecurity.com/content/security-testing-and-penetration-testing-products-and-services>
- [12] <http://www.infoworld.com/d/security-central/product-review-core-impact-penetration-tester-goes-phishing-390>
- [13] http://searchsecurity.techtarget.com/news/article/0,289142,sid14_gci1516321,00.html
- [14] <http://www.coresecurity.com/content/Professional-Services>
- [15] http://www.securitypark.co.uk/security_article263506.html
- [16] <http://www.coresecurity.com/content/corelabs-advisories>
- [17] http://www.computerworld.com/s/article/9176373/Security_firm_reveals_Microsoft_s_silent_patches?taxonomyId=17&pageNumber=1
- [18] <http://www.coresecurity.com/content/microsoft-windows-createwindow-function-callback-bug>
- [19] <http://www.coresecurity.com/content/CORE-2010-0407-Excel-PivotTable-CDR-overflow>
- [20] <http://www.coresecurity.com/content/hp-nnm-ovjava locale-buffer-overflow>

External links

- Corporate Website (<http://www.coresecurity.com>)
- Core Security Blog (<http://blog.coresecurity.com>)

Corner case

A **corner case** (or *pathological case*) is a problem or situation that occurs only outside of normal operating parameters – specifically one that manifests itself when multiple environmental variables or conditions are simultaneously at extreme levels, even though each parameter is within the specified range for that parameter.

For example, a loudspeaker might distort audio, but only when played at maximum volume, maximum bass, and in a high-humidity environment. Or a computer server may be unreliable, but only with the maximum complement of 64 processors, 512 GB of memory, and 10,000 signed-on users.

Contrast a corner case with an edge case, an issue that occurs only at a (single) maximum or minimum parameter. For example, a speaker that distorts audio at maximum volume, even in the absence of other extreme settings or conditions.

Corner cases are part of an engineer's lexicon – especially an engineer involved in testing or debugging a complex system. Corner cases are often harder and more expensive to reproduce, test, and optimize because they require maximal configurations in multiple dimensions. They are frequently less-tested, given the belief that few product users will, in practice, exercise the product at multiple simultaneous maximum settings. Expert users of systems therefore routinely find corner case anomalies, and in many of these, errors.

The term "corner case" comes about by physical analogy with "edge case". Where an edge case involves pushing one variable to a minimum or maximum, putting us at the "edge" of the configuration space, a corner case involves doing so with multiple variables, which would put us at a "corner" of a multidimensional configuration space.

Daikon (system)

For other uses, see Daikon (disambiguation)

Daikon is a computer program that detects likely invariants of programs. An invariant is a condition that always holds true at certain points in the program. It is mainly used^[1] for debugging programs in late development, or checking modifications to existing code.

Daikon can detect properties in C, C++, Java, Perl, and IOA programs, as well as spreadsheet files or other data sources. Daikon is easy to extend and is free software^[2]

External links

- Daikon^[3] Official home site
- Dynamically Discovering Likely Program Invariants^[4], Michael D. Ernst PhD. Thesis (using Daikon)

References

[1] Dynamically Discovering Likely Program Invariants (<http://groups.csail.mit.edu/pag/pubs/invariants-ernst-phdthesis-abstract.html>)

[2] (http://groups.csail.mit.edu/pag/daikon/download/doc/daikon_manual_html/daikon_10.html#SEC140) Daikon license

[3] <http://groups.csail.mit.edu/pag/daikon/>

[4] <http://www.cs.washington.edu/homes/mernst/pubs/invariants-ernst-phdthesis.pdf>

Data-driven testing

Data-driven testing (DDT) is a term used in the testing of computer software to describe testing done using a table of conditions directly as test inputs and verifiable outputs as well as the process where test environment settings and control are not hard-coded. In the simplest form the tester supplies the inputs from a row in the table and expects the outputs which occur in the same row. The table typically contains values which correspond to boundary or partition input spaces. In the control methodology, test configuration is "read" from a database.

Introduction

In the testing of software or programs, several methodologies are available for implementing this testing. Each of these methods co-exist because they differ in the effort required to create and subsequently maintain. The advantage of Data-driven testing is the ease to add additional inputs to the table when new partitions are discovered or added to the product or System Under Test. The cost aspect makes DDT cheap for automation but expensive for manual testing. One could confuse DDT with Table-driven testing, which this article needs to separate more clearly in future.

Methodology Overview

- **Data-driven testing** is the creation of test scripts to run together with their related data sets in a framework. The framework provides re-usable test logic to reduce maintenance and improve test coverage. Input and result (test criteria) data values can be stored in one or more central data sources or databases, the actual format and organisation can be implementation specific.

The data comprises variables used for both input values and output verification values. In advanced (mature) automation environments data can be harvested from a running system using a purpose-built custom tool or sniffer, the DDT framework thus performs playback of harvested data producing a powerful automated regression testing tool. Navigation through the program, reading of the data sources, and logging of test status and information are all coded in the test script.

Data Driven

Anything that has a potential to change (also called "Variability" and includes such as environment, end points, test data and locations, etc), is separated out from the test logic (scripts) and moved into an 'external asset'. This can be a configuration or test dataset. The logic executed in the script is dictated by the data values.

- **Keyword-driven testing** is similar except that the test case is contained in the set of data values and not embedded or "hard-coded" in the test script itself. The script is simply a "driver" (or delivery mechanism) for the data that is held in the data source

The databases used for data-driven testing can include:-

- datapools
 - ODBC source's
 - csv files
 - Excel files
 - DAO objects
 - ADO objects, etc.
-

References

- Carl Nagle: *Test Automation Frameworks* (<http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm>), Software Automation Framework Support on SourceForge (<http://safsdev.sourceforge.net/Default.htm>)

Decision table

Decision tables are a precise yet compact way to model complicated logic.^[1]

Decision tables, like flowcharts and if-then-else and switch-case statements, associate conditions with actions to perform, but in many cases do so in a more elegant way.

In the 1960s and 1970s a range of "decision table based" languages such as Filetab were popular for business programming.

Structure

The four quadrants

Conditions	Condition alternatives
Actions	Action entries

Each decision corresponds to a variable, relation or predicate whose possible values are listed among the condition alternatives. Each action is a procedure or operation to perform, and the entries specify whether (or in what order) the action is to be performed for the set of condition alternatives the entry corresponds to. Many decision tables include in their condition alternatives the don't care symbol, a hyphen. Using don't cares can simplify decision tables, especially when a given condition has little influence on the actions to be performed. In some cases, entire conditions thought to be important initially are found to be irrelevant when none of the conditions influence which actions are performed.

Aside from the basic four quadrant structure, decision tables vary widely in the way the condition alternatives and action entries are represented.^{[2] [3]} Some decision tables use simple true/false values to represent the alternatives to a condition (akin to if-then-else), other tables may use numbered alternatives (akin to switch-case), and some tables even use fuzzy logic or probabilistic representations for condition alternatives.^[4] In a similar way, action entries can simply represent whether an action is to be performed (check the actions to perform), or in more advanced decision tables, the sequencing of actions to perform (number the actions to perform).

Example

The limited-entry decision table is the simplest to describe. The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in a given column are to be performed.

A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients.

The following is a **balanced decision table**.

Printer troubleshooter

		Rules							
		Y	Y	Y	Y	N	N	N	N
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable		X						
	Check the printer-computer cable	X	X						
	Ensure printer software is installed	X	X	X	X				
	Check/replace ink	X	X		X	X			
	Check for paper jam		X	X					

Of course, this is just a simple example (and it does not necessarily correspond to the reality of printer troubleshooting), but even so, it demonstrates how decision tables can scale to several conditions with many possibilities.

Software engineering benefits

Decision tables, especially when coupled with the use of a domain-specific language, allow developers and policy experts to work from the same information, the decision tables themselves.

Tools to render nested if statements from traditional programming languages into decision tables can also be used as a debugging tool^{[5] [6]}

Decision tables have proven to be easier to understand and review than code, and have been used extensively and successfully to produce specifications for complex systems.^[7]

Program embedded decision tables

Decision tables can be, and often are, embedded within computer programs and used to 'drive' the logic of the program. A simple example might be a lookup table containing a range of possible input values and a function pointer to the section of code to process that input.

Static decision table

Input	Function Pointer
'1'	Function 1 (initialize)
'2'	Function 2 (process 2)
'9'	Function 9 (terminate)

Multiple conditions can be coded for in similar manner to encapsulate the entire program logic in the form of an 'executable' decision table or control table.

References

- [1] "A History of Decision Tables" (<http://www.catalyst.com/products/logicgem/overview.html>), www.catalyst.com
- [2] (http://web.sxu.edu/rogers/sys/decision_tables.html)
- [3] (<http://www.cems.uwe.ac.uk/jharney/table.html>)
- [4] Wets, Geert; Witlox, Frank; Timmermans, Harry; Vanthienen, Jan (1996), "Locational choice modelling using fuzzy decision tables" (<http://repository.tue.nl/672953>), Biennial Conference of the North American Fuzzy Information Processing Society, Berkeley, CA: IEEE, pp. 80–84,
- [5] "A Real CCIDE Example" (<http://twysf.users.sourceforge.net/example.shtml>)
- [6] Experience With The Cope Decision Table Processor (<http://www.cs.adelaide.edu.au/~dwyer/Cope.html>)
- [7] Udo W. Pooch, "Translation of Decision Tables," ACM Computing Surveys, Volume 6, Issue 2 (June 1974) Pages: 125–151
ISSN:0360-0300

Further reading

- Dwyer, B. and Hutchings, K. (1977) "Flowchart Optimisation in Cope, a Multi-Choice Decision Table" Aust. Comp. J. Vol. 9 No. 3 p. 92 (Sep. 1977).
- Fisher, D.L. (1966) "Data, Documentation and Decision Tables" Comm ACM Vol. 9 No. 1 (Jan. 1966) p. 26–31.
- General Electric Company (1962) GE-225 TABSOL reference manual and GF-224 TABSOL application manual CPB-147B (June 1962).
- Grindley, C.B.B. (1968) "The Use of Decision Tables within Systematics" Comp. J. Vol. 11 No. 2 p. 128 (Aug. 1968).
- Jackson, M.A. (1975) Principles of Program Design Academic Press
- Myers, H.J. (1972) "Compiling Optimised Code from Decision Tables" IBM J. Res. & Development (Sept. 1972) p. 489–503.
- Pollack, S.L. (1962) "DETAB-X: An improved business-oriented computer language" Rand Corp. Memo RM-3273-PR (August 1962)
- Schumacher, H. and Sevcik, K.C. (1976) "The Synthetic Approach to Decision Table Conversion" Comm. ACM Vol. 19 No. 6 (June 1976) p. 343–351

Decision-to-decision path

A decision-to-decision path, or DD-Path, is a path of execution (usually through a graph representing a program, such as a flow-chart) that does not include any conditional nodes. That is, it is the path of execution between two decisions.

DD (decision-decision) path is a path of nodes in a directed graph. A chain is a path in which:

- Initial and terminal nodes are distinct
- All interior nodes have in-degree = 1 and out-degree = 1

A DD-path is a chain in a program graph such that:

- It consists of a single node with in-degree = 0 (initial node)
- It consists of a single node with out-degree = 0 (terminal node)
- It consists of a single node with in-deg ≥ 2 or out-deg ≥ 2
- It consists of a single node with in-deg = 1 and out-deg = 1
- It is a maximal chain of length ≥ 1 .

Design predicates

Design predicates are a method invented by Thomas McCabe, to quantify the complexity of the integration of two units of software. Each of the four types of design predicates have an associated integration complexity rating. For pieces of code that apply more than one design predicate, integration complexity ratings can be combined.

The sum of the integration complexity for a unit of code, plus one, is the maximum number of test cases necessary to exercise the integration fully. Though a test engineer can typically reduce this by covering as many previously uncovered design predicates as possible with each new test. Also, some combinations of design predicates might be logically impossible.

Types of Calls

Unconditional Call

Unit A always calls unit B. This has an integration complexity of 0. For example:

```
unitA::functionA() {
    unitB->functionB();
}
```

Conditional Call

Unit A may or may not call unit B. This integration has a complexity of 1, and needs two tests: one that calls B, and one that doesn't.

```
unitA::functionA() {
    if (condition)
        unitB->functionB();
}
```


Mutually Exclusive Conditional Call

This is like a programming language's switch statement. Unit A calls exactly one of several possible units. Integration complexity is $n - 1$, where n is the number of possible units to call.

```
unitA::functionA() {
    switch (condition) {
        case 1:
            unitB->functionB();
            break;
        case 2:
            unitC->functionC();
            break;
        ...
        default:
            unitN->functionN();
            break;
    }
}
```

Iterative Call

In an iterative call, unit A calls unit B at least once, but maybe more. This integration has a complexity of 1. It also requires two tests: one that calls unit B once, and one test that calls it more than once.

```
unitA::functionA() {
    do {
        unitB->functionB();
    } while (condition);
}
```

Combining Calls

Any particular integration can combine several types of calls. For example, unit A may or may not call unit B; and if it does, it can call it one or more times. This integration combines a conditional call, with its integration complexity of 1, and an iterative call, with its integration complexity of 1. The combined integration complexity totals 2.

```
unitA::functionA() {
    if (someNumber > 0) {
        for ( i = 0 ; i < someNumber ; i++ ) {
            unitB->functionB();
        }
    }
}
```

Since the number of necessary tests is the total integration complexity plus one, this integration would require 3 tests. In one, where `someNumber` isn't greater than 0, unit B isn't called. In another, where `someNumber` is 1, unit B is called once. And in the final, `someNumber` is greater than 1, unit B is called more than once.

Development, testing, acceptance and production

The acronym DTAP is short for Development, Testing, Acceptance and Production^{[1] [2]}. It is a rather common acronym in ICT describing the steps taken during software development.

This is the sequence:

1. The program or component is developed on a **Development** system. This Development environment might have no testing capabilities.
2. Once the developer thinks it is ready, the product is copied to a **Test** environment, to verify it works as expected. This test environment is supposedly standardized and in close alignment with the target environment.
3. If the test is successful, the product is copied to an **Acceptance** test environment. During the Acceptance test, the customer will test the product in this environment to verify whether it meets their expectations.
4. If the customer accepts the product, it is deployed to **Production** environment, making it available to all users of the system.


The set of environments used for a DTAP cycle is often called a DTAP street.

References

- [1] Evans, Cal (2009) *Professional Programming: DTAP – Part 1: What is DTAP?* (<http://www.phparch.com/2009/07/27/professional-programming-dtap-part-1-what-is-dtap/>)
 - [2] Wiggers, Steef-Jan (2009) *DTAP Strategy: Pricing and Licensing* (<http://soa-thoughts.blogspot.com/2009/03/dtap-strategy-pricing-and-licensing.html>)
-

DeviceAnywhere

DeviceAnywhere

	
Type	Private
Industry	Enterprise Software Software Testing Quality Assurance
Founded	2003
Headquarters	San Mateo, California, USA
Area served	Worldwide
Key people	Faraz Syed, CEO David J. Marsyla, CTO Chris Callahan, SVP Global Sales Mark Dirsa, CFO Rachel Obstler, VP Product Management Leila Modarres, VP Marketing
Products	Test Automation for Smart Devices MonitorAnywhere Test Center
Employees	200 (2010)
Website	www.deviceanywhere.com ^[1]

DeviceAnywhere provides a service for testing and monitoring the functionality, usability, performance and availability of mobile apps and websites^{[2] [3] [4]}.

Products and services

Enterprises

- Test Automation for Smart Devices ^[5] is a SaaS enterprise software service that automates testing of mobile apps and websites^[6]. The product aims to ensure reliable, repeatable and reportable results for any mobile app, on any device, operating system and network. Using the service, engineers can perform unit testing on real devices. Similarly, testers are able to troubleshoot field issues and perform the following kinds of tests: UAT; BAT; Functional Testing; Regression Testing; and Integration Testing.
- MonitorAnywhere ^[7] is a SaaS platform targeted at enterprises, which provides monitoring of mobile services to detect potential problems early and ensure quality of service^[8].

Developers

- Test Center^[9] is a cloud-based service designed for manual testing of mobile apps. The service allows developers to remotely interact with and control a device's functions including pressing handset buttons, viewing LCD displays, listening to ringers and tones, playing videos, tapping and swiping touch screens, muting, powering on/off, increasing volume and other functions as if they were holding the physical device in their own hands.

Key Partnerships

HP

DeviceAnywhere provides an integrated solution for HP's Functional Testing (QTP)^[10] and HP Quality Center (QC)^[11]. The add-in allows developers to run complex test scenarios from QTP/QC in DeviceAnywhere Test Center and DeviceAnywhere Test Automation for Smart Devices and receive the results back into QC/QTP.

IBM

The DeviceAnywhere and IBM Rational Quality Manager integration allows companies using RQM as their test management system to test their products on real mobile devices^[12]. All results (script steps, test pass/fail results) are tracked by RQM and are available through the RQM interface.

History

The company was founded in April 2003 by CEO Faraz Syed and CTO David J. Marsyla, who jointly created the first remote access service for mobile devices. It was backed by Motorola Ventures and France Telecom's venture subsidiary, Innovacom.

- July 2004: First production release of DeviceAnywhere product
- December 2004: Partnership agreement with Motorola announced
- April 2005: First round of funding completed with Innovacom as lead investor
- November 2005: Expansion to EMEA market announced
- December 2006: Second round of funding completed with Motorola Ventures as lead investor
- February 2007: Office opened in London
- February 2007: DeviceAnywhere awarded Mobile Monday Global Peer Award for Community Favorite
- March 2007: DeviceAnywhere awarded Frost & Sullivan Mobility Award for Best User Experience Management
- March 2007: DeviceAnywhere reaches 1000 customer landmark
- May 2007: DeviceAnywhere announces partnership with Sprint
- April 2008: FierceWireless names DeviceAnywhere as one of its "Fierce 15"
- January 2009: DeviceAnywhere named top mobile banking development tool of 2008
- June 2009: Technology partnership with IBM announced
- July 2009: Facebook named as DeviceAnywhere customer
- August 2009: DeviceAnywhere named to Inc 500 list of fastest-growing private companies
- September 2009: DeviceAnywhere named to Software 500 list of fastest-growing software companies
- September 2009: DeviceAnywhere names new mobile healthcare customers
- October 2009: DeviceAnywhere named 16th fastest-growing company in Silicon Valley
- March 2010: DeviceAnywhere releases first "Mobile Metrics" report of mobile development trends
- September 2010: DeviceAnywhere releases first European "Mobile Metrics" report of mobile development trends
- October 2010: DeviceAnywhere launches Test Automation for Smart Devices to enterprise market
- October 2010: DeviceAnywhere CEO addresses CTIA on challenges of managing remote workforces
- January 2011: DeviceAnywhere launches first package for tablet devices
- February 2011: Alcatel-Lucent announces technology partnership with DeviceAnywhere

- March 2011: Test Automation for Smart Devices wins Mobility Award for enterprise mobile software innovation

References

- [1] <http://www.deviceanywhere.com/>
- [2] <http://blog.connectedplanetonline.com/unfiltered/2011/03/31/things-you-wanted-to-know-about-smartphone-development-but-because-you-dont-own-testing-labs-couldnt-ask/>
- [3] http://www.cio.com/article/631815/Emerging_Tech_Disaster_Proof_Your_Mobile_App_Before_Rollout
- [4] <http://www.fiercewireless.com/nextgenspotlight/story/cross-platform-apps-reflect-new-devices-higher-expectations>
- [5] <http://www.deviceanywhere.com/mobile-application-testing-smart-devices.html>
- [6] <http://www.networkworld.com/news/2010/100610-deviceanywhere-automated-testing-smartphones.html>
- [7] <http://www.deviceanywhere.com/mobile-application-testing-monitoring.html>
- [8] <http://www.gomonews.com/deviceanywhere-keeps-postemobile-mobile-services-ticking/>
- [9] <http://www.deviceanywhere.com/mobile-application-testing.html>
- [10] http://en.wikipedia.org/wiki/HP_QuickTest_Professional
- [11] http://en.wikipedia.org/wiki/HP_Quality_Center
- [12] https://www.ibm.com/developerworks/mydeveloperworks/blogs/social-media-marketing/entry/mobile_application_developers_test_in_the_wild_with_deviceanywhere?lang=en

External links

- DeviceAnywhere website (<http://www.deviceanywhere.com/>)
- DeviceAnywhere CEO blog (<http://deviceanywhere.typepad.com/>)
- DeviceAnywhere Test Center blog (<http://mobileapplicationtestingtimes.wordpress.com/>)
- DeviceAnywhere on Twitter (<http://twitter.com/DevAnywhere>)

Dry run (testing)

A **dry run** is a testing process where the effects of a possible failure are intentionally mitigated. For example, an aerospace company may conduct a "dry run" test of a jet's new pilot ejection seat while the jet is parked on the ground, rather than while it is in flight.

In computer programming, a dry run is a mental run of a computer program, where the computer programmer examines the source code one step at a time and determines what it will do when run. In theoretical computer science, a dry run is a mental run of an algorithm, sometimes expressed in pseudocode, where the computer scientist examines the algorithm's procedures one step at a time. In both uses, the dry run is frequently assisted by a table (on a computer screen or on paper) with the program or algorithm's variables on the top.

The usage of "dry run" in acceptance procedures (for example in the so called FAT = Factory Acceptance Testing) is meant as following: the factory - which is a subcontractor - must perform a complete test of the system it has to deliver *before* the actual acceptance from the contractor side.

External links

- World Wide Words: Dry Run ^[1]

References

- [1] <http://www.worldwidewords.org/qa/qa-dry1.htm>
-

Dynamic program analysis

Dynamic program analysis is the analysis of computer software that is performed by executing programs built from that software system on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior. Use of software testing techniques such as code coverage helps ensure that an adequate slice of the program's set of possible behaviors has been observed. Also, care must be taken to minimize the effect that instrumentation has on the execution (including temporal properties) of the target program. Inadequate testing can lead to catastrophic failures similar to the maiden flight of the Ariane 5 rocket launcher where dynamic execution errors (run time error) resulted in the destruction of the vehicle^[1].

Examples of Tools

- Avalanche is an open source tool that generates input data demonstrating crashes in the analysed program.
- BoundsChecker: Memory error detection for Windows based applications. Part of Micro Focus DevPartner.
- ClearSQL: is a review and quality control and a code illustration tool for PL/SQL.
- Daikon (system) is an implementation of dynamic invariant detection. Daikon runs a program, observes the values that the program computes, and then reports properties that were true over the observed executions, and thus likely true over all executions.
- Dmalloc, library for checking memory allocation and leaks. Software must be recompiled, and all files must include the special C header file dmalloc.h.
- DynInst is a runtime code-patching library that is useful in developing dynamic program analysis probes and applying them to compiled executables. Dyninst does not require source code or recompilation in general, however, non-stripped executables and executables with debugging symbols are easier to instrument.
- HP Security Suite is a suite of Tools at various stages of development. QAInspect and WebInspect are generally considered Dynamic Analysis Tools, while DevInspect is considered a static code analysis tool.
- IBM Rational AppScan is a suite of application security solutions targeted for different stages of the development lifecycle. The suite includes two main dynamic analysis products - IBM Rational AppScan Standard Edition, and IBM Rational AppScan Enterprise Edition. In addition, the suite includes IBM Rational AppScan Source Edition - a static analysis tool.
- Intel Thread Checker is a runtime threading error analysis tool which can detect potential data races and deadlocks in multithreaded Windows or Linux applications.
- Intel Parallel Inspector performs run time threading and memory error analysis in Windows.
- Parasoft Insure++ is runtime memory analysis and error detection tool. Its Inuse component provides a graphical view of memory allocations over time, with specific visibility into overall heap usage, block allocations, possible outstanding leaks, etc.
- Parasoft Jtest uses runtime error detection to expose defects such as race conditions, exceptions, resource & memory leaks, and security attack vulnerabilities.
- Polyspace uses abstract interpretation to detect and prove the absence of certain run time errors in source code.
- Purify: mainly memory corruption detection and memory leak detection.
- Valgrind runs programs on a virtual processor and can detect memory errors (e.g., misuse of malloc and free) and race conditions in multithread programs.
- VB Watch injects dynamic analysis code into Visual Basic programs to monitor their performance, call stack, execution trace, instantiated objects, variables and code coverage.

Most performance analysis tools use dynamic program analysis techniques.

Historical examples

- IBM OLIVER (CICS interactive test/debug): CICS application error detection including storage violations using an instruction Set Simulator to detect most CICS errors interactively
- SIMON (Batch Interactive test/debug) interactive batch program analyzer and test/debug using an instruction Set Simulator
- SIMMON: IBM internal instruction Set Simulator used for testing operating system components, utilities and I/O processors

References

- [1] Dowson, M. (March 1997). "The Ariane 5 Software Failure". *Software Engineering Notes* **22** (2): 84. doi:10.1145/251880.251992.

Dynamic testing

Dynamic testing (or dynamic analysis) is a term used in software engineering to describe the testing of the dynamic behavior of code. That is, dynamic analysis refers to the examination of the physical response from the system to variables that are not constant and change with time. In dynamic testing the software must actually be compiled and run; Actually Dynamic Testing involves working with the software, giving input values and checking if the output is as expected. These are the Validation activities. Unit Tests, Integration Tests, System Tests and Acceptance Tests are few of the Dynamic Testing methodologies. Dynamic testing means testing based on specific test cases by execution of the test object or running programs.

Dynamic testing is used to test software through executing it. This is in contrast to Static testing.

References

- G.J. Myers, The Art of Software Testing, John Wiley and Sons, New York, New York, 1979.

External links

- Dynamic software testing of MPI applications with umpire ^[1]

References

- [1] <http://portal.acm.org/citation.cfm?id=370049.370462>
-

Edge case

An **edge case** is a problem or situation that occurs only at an extreme (maximum or minimum) operating parameter.

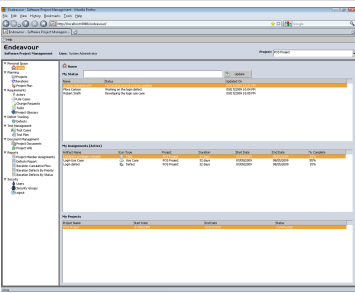
For example, a stereo speaker might distort audio when played at its maximum rated volume, even in the absence of other extreme settings or conditions.

An edge case can be expected or unexpected. In engineering, the process of planning for and gracefully addressing edge cases can be a significant task, and one that may be overlooked or underestimated. Non-trivial edge cases can result in a failure of the object being engineered that may not have been imagined during the design phase or anticipated as possible during normal use. For this reason, attempts to formalize good engineering practices often incorporate information about dealing with edge cases.

Endeavour Software Project Management

Endeavour Software Project Management

Endeavour



Endeavour's Home Page

Developer(s)	Ezequiel Cuellar
Stable release	1.25 / May 1, 2011
Written in	Java
Operating system	Cross-platform
Type	Project management software
License	GPL (free software)
Website	endeavour-mgmt.sourceforge.net ^[1]

Endeavour Software Project Management is an open source solution to manage large-scale enterprise software projects in an iterative and incremental development process.

History

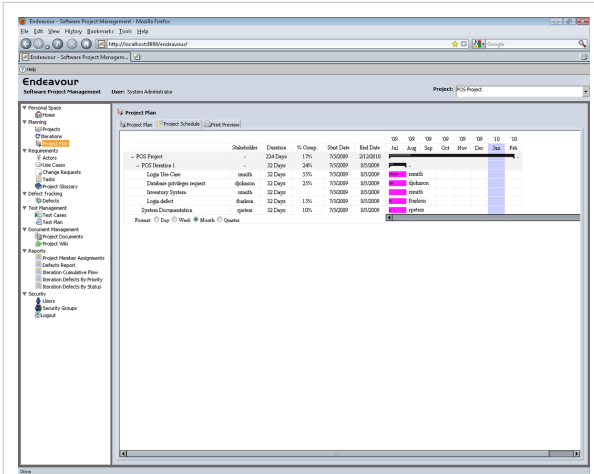
Endeavour Software Project Management was founded in September 2008 with the intention to develop a solution for replacing expensive and complex project management systems that is easy to use, intuitive, and realistic by eliminating features considered unnecessary.

In September 2009 the project was registered in SourceForge, and in April 2010 the project was included in SourceForge's blog with an average of 210 weekly downloads.

Features

The major features include support for the following software artifacts^[2]:

- Projects
- Use Cases
- Iterations



Endeavour's Project Plan Gantt Chart

- Project Plans
- Change Requests
- Defect Tracking
- Test Cases
- Test Plans
- Task
- Actors
- Document Management
- Project Glossary
- Project Wiki
- Developer Management
- Reports (Assignments, Defects, Cumulative Flow)
- SVN Browser Integration with svenson
- CI Integration with Hudson
- Email notifications
- Fully Internationalizable

System requirements

Endeavour Software Project Management can be deployed in any Java EE-compliant application server and any relational database running under a variety of different operating systems. Its cross-browser capability allows it to run in most popular web browsers.

Usage

- Software Project Management
- Iterative and Incremental development
- Use Case Driven
- Issue tracking
- Test Case Management Software
- Integrated Wiki

References

- [1] <http://sourceforge.net/projects/endeavour-mgmt/>
[2] [http://sourceforge.net/projects/endeavour-mgmt/Endeavour en SourceForge](http://sourceforge.net/projects/endeavour-mgmt/Endeavour%20en%20SourceForge)

Notes

- Lee Schlesinger. Social media specialist at SourceForge.net blog post about Endeavour Software Project Management (<http://sourceforge.net/blog/endeavour-to-improve-your-development-process/>)
- <http://sourceforge.net/projects/endeavour-mgmt/reviews> (<http://sourceforge.net/projects/endeavour-mgmt/reviews/>)
- <http://www.softpedia.com/get/Programming/Coding-languages-Compilers/Endeavour-Software-Project-Management.shtml> (<http://www.softpedia.com/get/Programming/Coding-languages-Compilers/Endeavour-Software-Project-Management.shtml>)
- <http://freshmeat.net/projects/endeavour-software-project-management> (<http://freshmeat.net/projects/endeavour-software-project-management>)
- <http://java.dzone.com/announcements/endeavour-software-project-2> (<http://java.dzone.com/announcements/endeavour-software-project-2>)

- <http://www.federalarchitect.com/2009/07/21/new-open-source-project-management-tool-for-large-scale-enterprise-systems/> (<http://www.federalarchitect.com/2009/07/21/new-open-source-project-management-tool-for-large-scale-enterprise-systems/>)

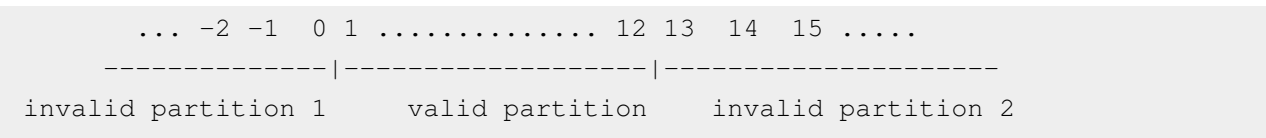
External links

- Endeavour Software Project Management (<http://sourceforge.net/projects/endeavour-mgmt/>) at SourceForge.net
- Download latest version (<http://sourceforge.net/projects/endeavour-mgmt/files/>)

Equivalence partitioning

Equivalence partitioning (also called **Equivalence Class Partitioning** or **ECP**^[1]) is a software testing technique that divides the input data of a software unit into partitions of data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once. This technique tries to define test cases that uncover classes of errors, thereby reducing the total number of test cases that must be developed.

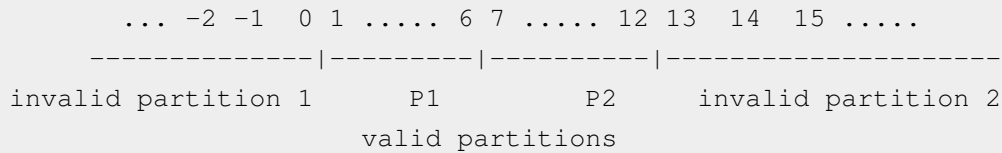
In rare cases equivalence partitioning is also applied to outputs of a software component, typically it is applied to the inputs of a tested component. The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object. An input has certain ranges which are valid and other ranges which are invalid. Invalid data here does not mean that the data is incorrect, it means that this data lies outside of specific partition. This may be best explained by the example of a function which takes a parameter "month". The valid range for the month is 1 to 12, representing January to December. This valid range is called a partition. In this example there are two further partitions of invalid ranges. The first invalid partition would be ≤ 0 and the second invalid partition would be ≥ 13 .



The testing theory related to equivalence partitioning says that only one test case of each partition is needed to evaluate the behaviour of the program for the related partition. In other words it is sufficient to select one test case out of each partition to check the behaviour of the program. To use more or even all test cases of a partition will not find new faults in the program. The values within one partition are considered to be "equivalent". Thus the number of test cases can be reduced considerably.

An additional effect of applying this technique is that you also find the so called "dirty" test cases. An inexperienced tester may be tempted to use as test cases the input data 1 to 12 for the month and forget to select some out of the invalid partitions. This would lead to a huge number of unnecessary test cases on the one hand, and a lack of test cases for the dirty ranges on the other hand.

The tendency is to relate equivalence partitioning to so called black box testing which is strictly checking a software component at its interface, without consideration of internal structures of the software. But having a closer look at the subject there are cases where it applies to grey box testing as well. Imagine an interface to a component which has a valid range between 1 and 12 like the example above. However internally the function may have a differentiation of values between 1 and 6 and the values between 7 and 12. Depending upon the input value the software internally will run through different paths to perform slightly different actions. Regarding the input and output interfaces to the component this difference will not be noticed, however in your grey-box testing you would like to make sure that both paths are examined. To achieve this it is necessary to introduce additional equivalence partitions which would not be needed for black-box testing. For this example this would be:



To check for the expected results you would need to evaluate some internal intermediate values rather than the output interface. It is not necessary that we should use multiple values from each partition. In the above scenario we can take -2 from invalid partition 1, 6 from valid partition P1, 7 from valid partition P2 and 15 from invalid partition 2.

Equivalence partitioning is not a stand alone method to determine test cases. It has to be supplemented by boundary value analysis. Having determined the partitions of possible inputs the method of boundary value analysis has to be applied to select the most effective test cases out of these partitions.

References

- The Testing Standards Working Party website ^[1]
- Parteg ^[2], a free test generation tool that is combining test path generation from UML state machines with equivalence class generation of input values.

[1] Burnstein, Ilene (2003), *Practical Software Testing*, Springer-Verlag, p. 623, ISBN 0-387-95131-8

[2] <http://parteg.sourceforge.net>

Error guessing

In software testing, **error guessing** is a test method in which test cases used to find bugs in programs are established based on experience in prior testing. The scope of test cases usually rely on the software tester involved, who uses past experience and intuition to determine what situations commonly cause software failure, or may cause errors to appear. Typical errors include divide by zero, null pointers, or invalid parameters.

Error guessing has no explicit rules for testing; ^[1] test cases can be designed depending on the situation, either drawing from functional documents or when an unexpected/undocumented error is found while testing operations.

References

- ISTQB Current Glossary (pdf) ^[2]

[1] "What is Error Guessing?" (<http://www.softwaretestingmentor.com/types-of-testing/error-guessing.php>). Software Testing Mentor. .

[2] <http://www.istqb.org/download/attachments/2326555/ISTQB+Glossary+of+Testing+Terms+2+1.pdf>

Exploratory testing

Exploratory testing is an approach to software testing that is concisely described as simultaneous learning, test design and test execution. Cem Kaner, who coined the term in 1983,^[1] now defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."^[2]

While the software is being tested, the tester learns things that together with experience and creativity generates new good tests to run. Exploratory testing is often thought of as a black box testing technique. Instead, those who have studied it consider it a test *approach* that can be applied to any test technique, at any stage in the development process. The key is not the test technique nor the item being tested or reviewed; the key is the cognitive engagement of the tester, and the tester's responsibility for managing his or her time.^[3]

History

Exploratory testing has always been performed by skilled testers. In the early 1990s, ad hoc was too often synonymous with sloppy and careless work. As a result, a group of test methodologists (now calling themselves the Context-Driven School) began using the term "exploratory" seeking to emphasize the dominant thought process involved in unscripted testing, and to begin to develop the practice into a teachable discipline. This new terminology was first published by Cem Kaner in his book *Testing Computer Software*^[1] and expanded upon in *Lessons Learned in Software Testing*.^[4] Exploratory testing can be as disciplined as any other intellectual activity.

Description

Exploratory testing seeks to find out how the software actually works, and to ask questions about how it will handle difficult and easy cases. The quality of the testing is dependent on the tester's skill of inventing test cases and finding defects. The more the tester knows about the product and different test methods, the better the testing will be.

To further explain, comparison can be made of freestyle exploratory testing to its antithesis scripted testing. In this activity test cases are designed in advance. This includes both the individual steps and the expected results. These tests are later performed by a tester who compares the actual result with the expected. When performing exploratory testing, expectations are open. Some results may be predicted and expected; others may not. The tester configures, operates, observes, and evaluates the product and its behaviour, critically investigating the result, and reporting information that seems likely to be a bug (which threatens the value of the product to some person) or an issue (which threatens the quality of the testing effort).

In reality, testing almost always is a combination of exploratory and scripted testing, but with a tendency towards either one, depending on context.

According to Cem Kaner & James Bach, exploratory testing is more a mindset or "...a way of thinking about testing" than a methodology.^[5] They also say that it crosses a continuum from slightly exploratory (slightly ambiguous or vaguely scripted testing) to highly exploratory (freestyle exploratory testing).^[6]

The documentation of exploratory testing ranges from documenting all tests performed to just documenting the bugs. During pair testing, two persons create test cases together; one performs them, and the other documents. Session-based testing is a method specifically designed to make exploratory testing auditable and measurable on a wider scale.

Exploratory testers often use tools, including screen capture or video tools as a record of the exploratory session, or tools to quickly help generate situations of interest, e.g. James Bach's Perclip.

Benefits and drawbacks

The main advantage of exploratory testing is that less preparation is needed, important bugs are found quickly, and at execution time, the approach tends to be more intellectually stimulating than execution of scripted tests.

Another major benefit is that testers can use deductive reasoning based on the results of previous results to guide their future testing on the fly. They do not have to complete a current series of scripted tests before focusing in on or moving on to exploring a more target rich environment. This also accelerates bug detection when used intelligently.

Another benefit is that, after initial testing, most bugs are discovered by some sort of exploratory testing. This can be demonstrated logically by stating, "Programs that pass certain tests tend to continue to pass the same tests and are more likely to fail other tests or scenarios that are yet to be explored."

Disadvantages are that tests invented and performed on the fly can't be reviewed in advance (and by that prevent errors in code and test cases), and that it can be difficult to show exactly which tests have been run.

Freestyle exploratory test ideas, when revisited, are unlikely to be performed in exactly the same manner, which can be an advantage if it is important to find new errors; or a disadvantage if it is more important to repeat specific details of the earlier tests. This can be controlled with specific instruction to the tester, or by preparing automated tests where feasible, appropriate, and necessary, and ideally as close to the unit level as possible.

Usage

Exploratory testing is particularly suitable if requirements and specifications are incomplete, or if there is lack of time.^[7]^[8] The approach can also be used to verify that previous testing has found the most important defects.^[7]

References

- [1] Kaner, Falk, and Nguyen, *Testing Computer Software (Second Edition)*, Van Nostrand Reinhold, New York, 1993. p. 6, 7-11.
- [2] Cem Kaner, *A Tutorial in Exploratory Testing* (<http://www.kaner.com/pdfs/QAIEExploring.pdf>), p. 36.
- [3] Cem Kaner, *A Tutorial in Exploratory Testing* (<http://www.kaner.com/pdfs/QAIEExploring.pdf>), p. 37-39, 40- .
- [4] Kaner, Cem; Bach, James; Pettichord, Bret (2001). *Lessons Learned in Software Testing*. John Wiley & Sons. ISBN 0471081124.
- [5] Cem Kaner, James Bach, *Exploratory & Risk Based Testing*, www.testingeducation.org (<http://www.testingeducation.org>), 2004, p. 10
- [6] Cem Kaner, James Bach, *Exploratory & Risk Based Testing*, www.testingeducation.org (<http://www.testingeducation.org>), 2004, p. 14
- [7] Bach, James (2003). "Exploratory Testing Explained" (<http://www.satisfice.com/articles/et-article.pdf>). [satisfice.com](http://www.satisfice.com). p. 7. . Retrieved October 23, 2010.
- [8] Kaner, Cem (2008). "A Tutorial in Exploratory Testing" (<http://www.kaner.com/pdfs/QAIEExploring.pdf>). [kaner.com](http://www.kaner.com). p. 37, 118. . Retrieved October 23, 2010.

External links

- James Bach, *Exploratory Testing Explained* (<http://www.satisfice.com/articles/et-article.pdf>)
- Cem Kaner, James Bach, *The Nature of Exploratory Testing* (<http://www.testingeducation.org/a/nature.pdf>), 2004
- Cem Kaner, James Bach, *The Seven Basic Principles of the Context-Driven School* (<http://www.context-driven-testing.com>)
- Jonathan Kohl, *Exploratory Testing: Finding the Music of Software Investigation* (<http://www.methodsandtools.com/archive/archive.php?id=65>), Kohl Concepts Inc., 2007
- Chris Agruss, Bob Johnson, *Ad Hoc Software Testing* (http://www.testingcraft.com/ad_hoc_testing.pdf)

Fagan inspection

Fagan inspection refers to a structured process of trying to find defects in development documents such as programming code, specifications, designs and others during various phases of the software development process. It is named after Michael Fagan who is credited with being the inventor of formal software inspections.

Definition

Fagan Inspection is a group review method used to evaluate output of a given process.

Fagan Inspection defines a process as a certain activity with a pre-specified entry and exit criteria. In every activity or operation for which entry and exit criteria are specified Fagan Inspections can be used to validate if the output of the process complies with the exit criteria specified for the process.

Examples of activities for which Fagan Inspection can be used are:

- Requirement specification
- Software/Information System architecture (for example DYA)
- Programming (for example for iterations in XP or DSDM)
- Software testing (for example when creating test scripts)

Usage

The software development process is a typical application of Fagan Inspection; software development process is a series of operations which will deliver a certain end product and consists of operations like requirements definition, design, coding up to testing and maintenance. As the costs to remedy a defect are up to 10-100 times less in the early operations compared to fixing a defect in the maintenance phase it is essential to find defects as close to the point of insertion as possible. This is done by inspecting the output of each operation and comparing that to the output requirements, or exit-criteria of that operation.

Criteria

Entry criteria are the criteria or requirements which must be met to enter a specific process^[1]. For example for Fagan inspections the high- and low-level documents must comply with specific entry-criteria before they can be used for a formal inspection process.

Exit criteria are the criteria or requirements which must be met to complete a specific process. For example for Fagan inspections the low-level document must comply with specific exit-criteria (as specified in the high-level document) before the development process can be taken to the next phase.

The exit-criteria are specified in a high-level document, which is then used as the standard to compare the operation result (low-level document) to during the inspections. Deviations of the low-level document from the requirements specified in the high-level document are called defects and can be categorized in Major Defects and Minor Defects.

Defects

According to M.E. Fagan, "A defect is an instance in which a requirement is not satisfied."^[1]

In the process of software inspection the defects which are found are categorized in two categories: major and minor defects (often many more categories are used). The defects which are incorrect or even missing functionality or specifications can be classified as major defects: the software will not function correctly when these defects are not being solved.

In contrast to major defects, minor defects do not threaten the correct functioning of the software, but are mostly small errors like spelling mistakes in documents or optical issues like incorrect positioning of controls in a program

interface.

Typical operations

In a typical Fagan inspection the inspection process consists of the following operations^[1]:

- Planning
 - Preparation of materials
 - Arranging of participants
 - Arranging of meeting place
- Overview
 - Group education of participants on the materials under review
 - Assignment of roles
- Preparation
 - The participants review the item to be inspected and supporting material to prepare for the meeting noting any questions or possible defects
 - The participants prepare their roles
- Inspection meeting
 - Actual finding of defect
- Rework
 - Rework is the step in software inspection in which the defects found during the inspection meeting are resolved by the author, designer or programmer. On the basis of the list of defects the low-level document is corrected until the requirements in the high-level document are met.
- Follow-up
 - In the follow-up phase of software inspections all defects found in the inspection meeting should be corrected (as they have been fixed in the rework phase). The moderator is responsible for verifying that this is indeed the case. He should verify if all defects are fixed and no new defects are inserted while trying to fix the initial defects. It is crucial that all defects are corrected as the costs of fixing them in a later phase of the project will be 10 to 100 times higher compared to the current costs.

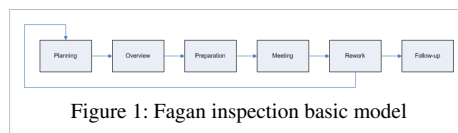


Figure 1: Fagan inspection basic model

Follow-up

In the follow-up phase of a Fagan Inspection, defects fixed in the rework phase should be verified. The moderator is usually responsible for verifying rework. Sometimes fixed work can be accepted without being verified, such as when the defect was trivial. In non-trivial cases, a full re-inspection is performed by the inspection team (not only the moderator).

If verification fails, go back to the rework process.

Roles

The participants of the inspection process are normally just members of the team that is performing the project. The participants fulfill different roles within the inspection process^{[2] [3]} :

- Author/Designer/Coder: the person who wrote the low-level document
- Reader: paraphrases the document
- Reviewers: reviews the document from a testing standpoint
- Moderator: responsible for the inspection session, functions as a coach

Benefits and results

By using inspections the number of errors in the final product can significantly decrease, creating a higher quality product. In the future the team will even be able to avoid errors as the inspection sessions give them insight in the most frequently made errors in both design and coding providing avoidance of error at the root of their occurrence. By continuously improving the inspection process these insights can even further be used [Fagan, 1986].

Together with the qualitative benefits mentioned above major "cost improvements" can be reached as the avoidance and earlier detection of errors will reduce the amount of resources needed for debugging in later phases of the project.

In practice very positive results have been reported by large corporations like IBM indicating that 80-90% of defects can be found and savings in resources up to 25% can be reached [Fagan, 1986]...

Improvements

Although the Fagan Inspection method has proved to be very effective, improvements have been suggested by multiple researchers. Genuchten for example has been researching the usage of an Electronic Meeting System (EMS) to improve the productivity of the meetings with positive results [Genuchten, 1997].

Other researchers propose the usage of software that keeps a database of detected errors and automatically scans program code for these common errors [Doolan,1992]. This again should result in improved productivity.

Example

In the diagram a very simple example is given of an inspection process in which a two-line piece of code is inspected on the basis on a high-level document with a single requirement.

As can be seen in the high-level document for this project is specified that in all software code produced variables should be declared 'strong typed'. On the basis of this requirement the low-level document is checked for defects. Unfortunately a defect is found on line 1, as a variable is not declared 'strong typed'. The defect found is then reported in the list of defects found and categorized according to the categorizations specified in the high-level document.

References

- [1] Fagan, M.E., Advances in Software Inspections, July 1986, IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, Page 744-751 (<http://www.mfagan.com/pdfs/aisi1986.pdf>)
- [2] M.E., Fagan (1976). "Design and Code inspections to reduce errors in program development". *IBM Systems Journal* **15** (3): pp. 182–211. (<http://www.mfagan.com/pdfs/ibmfagan.pdf>)
- [3] Eickelmann, Nancy S, Ruffolo, Francesca, Baik, Jongmoon, Anant, A, 2003 An Empirical Study of Modifying the Fagan Inspection Process and the Resulting Main Effects and Interaction Effects Among Defects Found, Effort Required, Rate of Preparation and Inspection, Number of Team Members and Product 1st Pass Quality, Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop
1. [Laitenberger, 1999] Laitenberger, O, DeBaud, J.M, 1999 An encompassing life cycle centric survey of software inspection, *Journal of Systems and Software* **50** (2000), Page 5-31
2. [So, 1995] So, S, Lim, Y, Cha, S.D., Kwon, Y.J, 1995 An Empirical Study on Software Error Detection: Voting, Instrumentation, and Fagan Inspection *, Proceedings of the 1995 Asia Pacific Software Engineering Conference (APSEC '95), Page 345-351
3. [Doolan, 1992] Doolan, E.P., 1992 Experience with Fagan's Inspection Method, *SOFTWARE—PRACTICE AND EXPERIENCE*, (FEBRUARY 1992) Vol. 22(2), Page 173–182
4. [Genuchten, 1997] Genuchten, M, Cornelissen, W, Van Dijk, C, 1997 Supporting Inspections with an Electronic Meeting System, *Journal of Management Information Systems*, Winter 1997-98/Volume 14, No. 3, Page 165-179

Fault injection

In software testing, **fault injection** is a technique for improving the coverage of a test by introducing faults to test code paths, in particular error handling code paths, that might otherwise rarely be followed. It is often used with stress testing and is widely considered to be an important part of developing robust software^[1]. Robustness testing^[2] (also known as Syntax Testing, Fuzzing or Fuzz testing) is a type of fault injection commonly used to test for vulnerabilities in communication interfaces such as protocols, command line parameters, or APIs.

The propagation of a fault through to an observable failure follows a well defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary. An error may cause further errors within the system boundary, therefore each new error acts as a fault, or it may propagate to the system boundary and be observable. When error states are observed at the system boundary they are termed failures. This mechanism is termed the fault-error-failure cycle^[3] and is a key mechanism in dependability.

History

The technique of fault injection dates back to the 1970s^[4] when it was first used to induce faults at a hardware level. This type of fault injection is called Hardware Implemented Fault Injection (HWIFI) and attempts to simulate hardware failures within a system. The first experiments in hardware fault injection involved nothing more than shorting connections on circuit boards and observing the effect on the system (bridging faults). It was used primarily as a test of the dependability of the hardware system. Later specialised hardware was developed to extend this technique, such as devices to bombard specific areas of a circuit board with heavy radiation. It was soon found that faults could be induced by software techniques and that aspects of this technique could be useful for assessing software systems. Collectively these techniques are known as Software Implemented Fault Injection (SWIFI).

Software Implemented fault injection

SWIFI techniques for software fault injection can be categorized into two types: compile-time injection and runtime injection.

Compile-time injection is an injection technique where source code is modified to inject simulated faults into a system. One method is called mutation testing which changes existing lines of code so that they contain faults. A simple example of this technique could be changing

```
a = a + 1
to
a = a - 1
```

Code mutation produces faults which are very similar to those unintentionally added by programmers.

A refinement of code mutation is *Code Insertion Fault Injection* which adds code, rather than modifies existing code. This is usually done through the use of perturbation functions which are simple functions which take an existing value and perturb it via some logic into another value, for example

```
int pFunc(int value) {
    return value + 20;
}
int main(int argc, char * argv[]) {
    int a = pFunc(aFunction(atoi(argv[1])));
    if (a > 20) {
        /* do something */
    } else {
        /* do something else */
    }
}
```

In this case pFunc is the perturbation function and it is applied to the return value of the function that has been called introducing a fault into the system.

Runtime Injection techniques use a software trigger to inject a fault into a running software system. Faults can be injected via a number of physical methods and triggers can be implemented in a number of ways, such as: Time Based triggers (When the timer reaches a specified time an interrupt is generated and the interrupt handler associated with the timer can inject the fault.); Interrupt Based Triggers (Hardware exceptions and software trap mechanisms are used to generate an interrupt at a specific place in the system code or on a particular event within the system, for instance access to a specific memory location).

Runtime injection techniques can use a number of different techniques to insert faults into a system via a trigger.

- Corrupting of memory space: This technique consists of corrupting RAM, processor registers, and I/O map.
- Syscall interposition techniques: This is concerned with the fault propagation from operating system kernel interfaces to executing systems software. This is done by intercepting operating system calls made by user-level software and injecting faults into them.
- Network Level fault injection: This technique is concerned with the corruption, loss or reordering of network packets at the network interface.

These techniques are often based around the debugging facilities provided by computer processor architectures.

Protocol software fault injection

Complex software systems, especially multi-vendor distributed systems based on open standards, perform input/output operations to exchange data via stateful, structured exchanges known as "protocols." One kind of fault injection that is particularly useful to test protocol implementations (a type of software code that has the unusual characteristic in that it cannot predict or control its input) is fuzzing. Fuzzing is an especially useful form of Black-box testing since the various invalid inputs that are submitted to the software system do not depend on, and are not created based on knowledge of, the details of the code running inside the system.

Fault injection tools

Although these types of faults can be injected by hand the possibility of introducing an unintended fault is high, so tools exist to parse a program automatically and insert faults.

Research tools

A number of SWIFI Tools have been developed and a selection of these tools is given here. Six commonly used fault injection tools are Ferrari, FTAPE, Doctor, Orchestra, Xception and Grid-FIT.

- MODIFI (MODEl-Implemented Fault Injection) is a fault injection tool for robustness evaluation of Simulink behavior models. It supports fault modelling in XML for implementation of domain-specific fault models.^[5]
- Ferrari (Fault and ERROR Automatic Real-time Injection) is based around software traps that inject errors into a system. The traps are activated by either a call to a specific memory location or a timeout. When a trap is called the handler injects a fault into the system. The faults can either be transient or permanent. Research conducted with Ferrari shows that error detection is dependent on the fault type and where the fault is inserted^[6].
- FTAPE (Fault Tolerance and Performance Evaluator) can inject faults, not only into memory and registers, but into disk accesses as well. This is achieved by inserting a special disk driver into the system that can inject faults into data sent and received from the disk unit. FTAPE also has a synthetic load unit that can simulate specific amounts of load for robustness testing purposes^[7].
- DOCTOR (IntegrateD Software Fault InJeCTiOn EnviRonment) allows injection of memory and register faults, as well as network communication faults. It uses a combination of time-out, trap and code modification. Time-out triggers inject transient memory faults and traps inject transient emulated hardware failures, such as register corruption. Code modification is used to inject permanent faults^[8].
- Orchestra is a script driven fault injector which is based around Network Level Fault Injection. Its primary use is the evaluation and validation of the fault-tolerance and timing characteristics of distributed protocols. Orchestra was initially developed for the Mach Operating System and uses certain features of this platform to compensate for latencies introduced by the fault injector. It has also been successfully ported to other operating systems^[9].
- Xception is designed to take advantage of the advanced debugging features available on many modern processors. It is written to require no modification of system source and no insertion of software traps, since the processor's exception handling capabilities trigger fault injection. These triggers are based around accesses to specific memory locations. Such accesses could be either for data or fetching instructions. It is therefore possible to accurately reproduce test runs because triggers can be tied to specific events, instead of timeouts^[10].
- Grid-FIT (Grid – Fault Injection Technology)^[11] is a dependability assessment method and tool for assessing Grid services by fault injection. Grid-FIT is derived from an earlier fault injector WS-FIT^[12] which was targeted towards Java Web Services implemented using Apache Axis transport. Grid-FIT utilises a novel fault injection mechanism that allows network level fault injection to be used to give a level of control similar to Code Insertion fault injection whilst being less invasive^[13].
- LFI (Library-level Fault Injector)^[14] is an automatic testing tool suite, used to simulate in a controlled testing environment, exceptional situations that programs need to handle at runtime but that are not easy to check via input testing alone. LFI automatically identifies the errors exposed by shared libraries, finds potentially buggy

error recovery code in program binaries and injects the desired faults at the boundary between shared libraries and applications.

Commercial tools

- ExhaustiF is a commercial software tool used for grey box testing based on software fault injection (SWIFI) to improve reliability of software intensive systems. The tool can be used during system integration and system testing phases of any software development lifecycle, complementing other testing tools as well. ExhaustiF is able to inject faults into both software and hardware. When injecting simulated faults in software, ExhaustiF offers the following fault types: Variable Corruption and Procedure Corruption. The catalogue for hardware fault injections includes faults in Memory (I/O, RAM) and CPU (Integer Unit, Floating Unit). There are different versions available for RTEMS/ERC32, RTEMS/Pentium, Linux/Pentium and MS-Windows/Pentium.^[15]
- Holodeck^[16] is a test tool developed by Security Innovation that uses fault injection to simulate real-world application and system errors for Windows applications and services. Holodeck customers include many major commercial software development companies, including Microsoft, Symantec, EMC and Adobe. It provides a controlled, repeatable environment in which to analyze and debug error-handling code and application attack surfaces for fragility and security testing. It simulates file and network fuzzing faults as well as a wide range of other resource, system and custom-defined faults. It analyzes code and recommends test plans and also performs function call logging, API interception, stress testing, code coverage analysis and many other application security assurance functions.
- Codenomicon Defensics^[17] is a blackbox test automation framework that does fault injection to more than 150 different interfaces including network protocols, API interfaces, files, and XML structures. The commercial product was launched in 2001, after five years of research at University of Oulu in the area of software fault injection. A thesis work explaining the used fuzzing principles was published by VTT, one of the PROTOS consortium members.^[18]
- The Mu Service Analyzer^[19] is a commercial service testing tool developed by Mu Dynamics^[20]. The Mu Service Analyzer performs black box and white box testing of services based on their exposed software interfaces, using denial-of-service simulations, service-level traffic variations (to generate invalid inputs) and the replay of known vulnerability triggers. All these techniques exercise input validation and error handling and are used in conjunction with valid protocol monitors and SNMP to characterize the effects of the test traffic on the software system. The Mu Service Analyzer allows users to establish and track system-level reliability, availability and security metrics for any exposed protocol implementation. The tool has been available in the market since 2005 by customers in the North America, Asia and Europe, especially in the critical markets of network operators (and their vendors) and Industrial control systems (including Critical infrastructure).
- Xception^[21] is a commercial software tool developed by Critical Software SA^[22] used for black box and white box testing based on software fault injection (SWIFI) and Scan Chain fault injection (SCIFI). Xception allows users to test the robustness of their systems or just part of them, allowing both Software fault injection and Hardware fault injection for a specific set of architectures. The tool has been used in the market since 1999 and has customers in the American, Asian and European markets, especially in the critical market of aerospace and the telecom market. The full Xception product family includes: a) The main Xception tool, a state-of-the-art leader in Software Implemented Fault Injection (SWIFI) technology; b) The Easy Fault Definition (EFD) and Xtract (Xception Analysis Tool) add-on tools; c) The extended Xception tool (eXception), with the fault injection extensions for Scan Chain and pin-level forcing.

Libraries

- `libfiu` ^[23] (Fault injection in userspace), C library to simulate faults in POSIX routines without modifying the source code. An API is included to simulate arbitrary faults at run-time at any point of the program.
- `TestApi` ^[24] is a shared-source API library, which provides facilities for fault injection testing as well as other testing types, data-structures and algorithms for .NET applications.

Application of fault injection

Fault injection can take many forms. In the testing of operating systems for example, fault injection is often performed by a *driver* (kernel-mode software) that intercepts *system calls* (calls into the kernel) and randomly returning a failure for some of the calls. This type of fault injection is useful for testing low level user mode software. For higher level software, various methods inject faults. In managed code, it is common to use instrumentation. Although fault injection can be undertaken by hand a number of fault injection tools exist to automate the process of fault injection ^[25].

Depending on the complexity of the API for the level where faults are injected, fault injection tests often must be carefully designed to minimise the number of false positives. Even a well designed fault injection test can sometimes produce situations that are impossible in the normal operation of the software. For example, imagine there are two API functions, `Commit` and `PrepareForCommit`, such that alone, each of these functions can possibly fail, but if `PrepareForCommit` is called and succeeds, a subsequent call to `Commit` is guaranteed to succeed. Now consider the following code:

```
error = PrepareForCommit();
if (error == SUCCESS) {
    error = Commit();
    assert(error == SUCCESS);
}
```

Often, it will be infeasible for the fault injection implementation to keep track of enough state to make the guarantee that the API functions make. In this example, a fault injection test of the above code might hit the `assert`, whereas this would never happen in normal operation.

References

- [1] J. Voas, "Fault Injection for the Masses," *Computer*, vol. 30, pp. 129–130, 1997.
- [2] Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security. 2001. (<http://www.vtt.fi/inf/pdf/publications/2001/P448.pdf>)
- [3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *Dependable and Secure Computing*, vol. 1, pp. 11–33, 2004.
- [4] J. V. Carreira, D. Costa, and S. J. G., "Fault Injection Spot-Checks Computer System Dependability," *IEEE Spectrum*, pp. 50–55, 1999.
- [5] Rickard Svenningsson, Jonny Vinter, Henrik Eriksson and Martin Torngren, "MODIFI: A MODEL-Implemented Fault Injection Tool," *Lecture Notes in Computer Science*, 2010, Volume 6351/2010, 210-222.
- [6] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Transactions on Computers*, vol. 44, pp. 248, 1995.
- [7] T. Tsai and R. Iyer, "FTAPE: A Fault Injection Tool to Measure Fault Tolerance," presented at Computing in aerospace, San Antonio; TX, 1995.
- [8] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An IntegrateD Software Fault InjeCTiOn EnviRonment for Distributed Real-time Systems," presented at International Computer Performance and Dependability Symposium, Erlangen; Germany, 1995.
- [9] S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations," presented at International Computer Performance and Dependability Symposium, Urbana-Champaign, USA, 1996.
- [10] J. V. Carreira, D. Costa, and S. J. G., "Fault Injection Spot-Checks Computer System Dependability," *IEEE Spectrum*, pp. 50–55, 1999.
- [11] Grid-FIT Web-site (<http://wiki.grid-fit.org/>)
- [12] N. Looker, B. Gwynne, J. Xu, and M. Munro, "An Ontology-Based Approach for Determining the Dependability of Service-Oriented Architectures," in the proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems, USA, 2005.

- [13] N. Looker, M. Munro, and J. Xu, "A Comparison of Network Level Fault Injection with Code Insertion," in the proceedings of the 29th IEEE International Computer Software and Applications Conference, Scotland, 2005.
- [14] LFI Website (<http://lfi.epfl.ch/>)
- [15] ExhaustiF SWIFI Tool Site (<http://www.exhaustif.es>)
- [16] Holodeck product overview (<http://www.securityinnovation.com/holodeck/index.shtml>)
- [17] Codenomicon Defensics product overview (<http://www.codenomicon.com/defensics/>)
- [18] Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security. 2001. (<http://www.vtt.fi/inf/pdf/publications/2001/P448.pdf>)
- [19] Mu Service Analyzer (<http://www.mudynamics.com/products/overview.html>)
- [20] Mu Dynamics, Inc. (<http://www.mudynamics.com/>)
- [21] Xception Web Site (<http://www.xception.org>)
- [22] Critical Software SA (<http://www.criticalsoftware.com>)
- [23] <http://blitiri.com.ar/p/libfiu/>
- [24] <http://testapi.codeplex.com>
- [25] N. Looker, M. Munro, and J. Xu, "Simulating Errors in Web Services," International Journal of Simulation Systems, Science & Technology, vol. 5, 2004.

External links

- Using Fault Injection to Test Software Recovery Code (<http://www.cs.colostate.edu/casi/REPORTS/Bieman95.pdf>) by Colorado Advanced Software Institute.
- Certitude Software from Certess Inc. (<http://www.certess.com/product/>)

Financial tester

The **financial testers** are the software testers who concern themselves with the way data flows through a transactional system, this data should have a start point and have a method by which to track the data through its different states (offers, rates, exchange, commissions, etc.). This testing should be carried out in a controlled environment, using this technique it is possible to pin point several areas that may have been overlooked due to the nature of the transactions that have been processed.

In general a Financial Tester should have a good understanding of financial information and understand reconciliation methodologies, as these are key in this role for data validation.

Framework for Integrated Test

Framework for Integrated Test, or "Fit", is an open-source tool for automated customer tests. It integrates the work of customers, analysts, testers, and developers.

Customers provide examples of how their software should work. Those examples are then connected to the software with programmer-written test fixtures and automatically checked for correctness. The customers' examples are formatted in tables and saved as HTML using ordinary business tools such as Microsoft Excel. When Fit checks the document, it creates a copy and colors the tables green, red, and yellow according to whether the software behaved as expected.

Fit was invented by Ward Cunningham in 2002. He created the initial Java version of Fit. As of June 2005, it has up-to-date versions for Java, C#, Python, Perl, PHP and Smalltalk.

Although Fit is an acronym, the word "Fit" came first, making it a backronym. Fit is sometimes italicized but should not be capitalized. In other words, "Fit" and "*Fit*" are appropriate usage, but "FIT" is not.

Fit includes a simple command-line tool for checking Fit documents. There are third-party front-ends available. Of these, FitNesse is the most popular. FitNesse is a complete IDE for Fit that uses a Wiki for its front end. As of June 2005, FitNesse had forked Fit, making it incompatible with newer versions of Fit, but plans were underway to re-merge with Fit.

References

- R Mugridge & W Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, Prentice Hall PTR (2005)

External links

- Fit website ^[1]
- FitNesse website ^[2]
- Concordion ^[3] - a Java testing framework similar to Fit
- PHPFIT is a PHP5 port of the Framework for Integrated Test (FIT) ^[4]

References

[1] <http://fit.c2.com>

[2] <http://fitnesse.org>

[3] <http://www.concordion.org>

[4] <http://developer.berlios.de/projects/phpfit/>

Functional testing

Functional testing is a type of black box testing that bases its test cases on the specifications of the software component under test. Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered (Not like in white-box testing).^[1]

Functional testing differs from system testing in that functional testing "*verifies*" a program by checking it against ... design document(s) or specification(s)", while system testing "*validates*" a program by checking it against the published user or system requirements"(Kaner, Falk, Nguyen 1999, p. 52).

Functional testing typically involves five steps:

1. The identification of functions that the software is expected to perform
2. The creation of input data based on the function's specifications
3. The determination of output based on the function's specifications
4. The execution of the test case
5. The comparison of actual and expected outputs

Notes

[1] Kaner, Falk, Nguyen. *Testing Computer Software*. Wiley Computer Publishing, 1999, p. 42. ISBN 0-471-35846-0.

Functionality assurance

In computers, **functionality assurance** is a form of continuous testing to assure a working system remains functional.

From a technology risk point of view, there are a number of long-term risks (difficult to envisage) that might result in unacceptable application functionality status. The functionality assurance model asserts that it is not acceptable to detect reduced functionality through user interaction and is cost beneficial both from a functionality and a risk management point of view to assure that the applications within scope operate at full functionality. There are many states that can produce reduced functionality, such as security updates to operating systems, internal system errors, changes to the external application context and even application updates. Functionality assurance is not performed with automated vulnerability scanning as such scans cannot detect introduced or undetected vulnerabilities.

Anomalous application states include:

- OS (Operating System) not functional and application 100% disabled
- OS partially functional and application partially disabled
- Application 100% disabled through internal fault
- Application partially disabled through internal fault
- OS or application vulnerability introduced

To perform effective functionality assurance, a two level approach is taken. Regressions test are undertaken by different areas, such as:

- From an OS point of view, tests to verify required functionality (OS build team).
 - From an application point of view, test to verify the application functionality (Application developers).
 - The regression tests should be layered and should focus on providing a system "green light" if all required functionality is present or if not, identify the subsystem that failed the tests.
 - Trouble shooting should be a separate programme (too long a piece of string to be contained in a programme like this and very dependent on maturity of software engineering team).
-

- Software programmers should provide "call-back" functionality so that system monitors can verify the application functionality.
- The operations management team develop regression tests to verify the status of the OS
- The operations management team schedule the automated running of these regression tests to verify that both the application and the OS is still providing the required functionality after security updates, patch updates etc.

Fuzz testing

Fuzz testing or **fuzzing** is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes or failing built-in code assertions. Fuzzing is commonly used to test for security problems in software or computer systems.

The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing.

There are two forms of fuzzing program; *mutation-based* and *generation-based*, which can be employed as white-, grey- or black-box testing.^[1] File formats and network protocols are the most common targets of testing, but any type of program input can be fuzzed. Interesting inputs include environment variables, keyboard and mouse events, and sequences of API calls. Even items not normally considered "input" can be fuzzed, such as the contents of databases, shared memory, or the precise interleaving of threads.

For the purpose of security, input that crosses a trust boundary is often the most interesting.^[2] For example, it is more important to fuzz code that handles the upload of a file by any user than it is to fuzz the code that parses a configuration file that is accessible only to a privileged user.

History

The term "fuzz" or "fuzzing" originates from a 1988 class project at the University of Wisconsin, taught by Professor Barton Miller. The assignment was titled "Operating System Utility Program Reliability - The Fuzz Generator".^[3] ^[4] The project developed a basic command-line fuzzer to test the reliability of Unix programs by bombarding them with random data until they crashed. The test was repeated in 1995, expanded to include testing of GUI-based tools (X Windows), network protocols, and system library APIs.^[1] Follow-on work included testing command- and GUI-based applications on both Windows and MacOS X.

One of the earliest examples of fuzzing dates from before 1983. "The Monkey" was a Macintosh application developed by Steve Capps prior to 1983. It used journaling hooks to feed random events into Mac programs, and was used to test for bugs in MacPaint.^[5]

Uses

Fuzz testing is often employed as a black-box testing methodology in large software projects where a budget exists to develop test tools. Fuzz testing is one of the techniques which offers a high benefit to cost ratio.

The technique can only provide a random sample of the system's behavior, and in many cases passing a fuzz test may only demonstrate that a piece of software can handle exceptions without crashing, rather than behaving correctly. This means fuzz testing is an assurance of overall quality, rather than a bug-finding tool, and not a substitute for exhaustive testing or formal methods.

As a gross measurement of reliability, fuzzing can suggest which parts of a program should get special attention, in the form of a code audit, application of static analysis, or partial rewrites.

Types of bugs

As well as testing for outright crashes, fuzz testing is used to find bugs such as assertion failures and memory leaks (when coupled with a memory debugger). The methodology is useful against large applications, where any bug affecting memory safety is likely to be a severe vulnerability.

Since fuzzing often generates invalid input it is used for testing error-handling routines, which are important for software that does not control its input. Simple fuzzing can be thought of as a way to automate negative testing.

Fuzzing can also find some types of "correctness" bugs. For example, it can be used to find incorrect-serialization bugs by complaining whenever a program's serializer emits something that the same program's parser rejects.^[6] It can also find unintentional differences between two versions of a program^[7] or between two implementations of the same specification.^[8]

Techniques

Fuzzing programs fall into two different categories. Mutation based fuzzers mutate existing data samples to create test data while generation based fuzzers define new test data based on models of the input.^[1]

The simplest form of fuzzing technique is sending a stream of random bits to software, either as command line options, randomly mutated protocol packets, or as events. This technique of random inputs still continues to be a powerful tool to find bugs in command-line applications, network protocols, and GUI-based applications and services. Another common technique that is easy to implement is mutating existing input (e.g. files from a test suite) by flipping bits at random or moving blocks of the file around. However, the most successful fuzzers have detailed understanding of the format or protocol being tested.

The understanding can be based on a specification. A specification-based fuzzer involves writing the entire array of specifications into the tool, and then using model-based test generation techniques in walking through the specifications and adding anomalies in the data contents, structures, messages, and sequences. This "smart fuzzing" technique is also known as robustness testing, syntax testing, grammar testing, and (input) fault injection.^[9] ^[10] ^[11] ^[12] The protocol awareness can also be created heuristically from examples using a tool such as Sequitur ^[13].^[14] These fuzzers can *generate* test cases from scratch, or they can *mutate* examples from test suites or real life. They can concentrate on *valid* or *invalid* input, with *mostly-valid* input tending to trigger the "deepest" error cases.

There are two limitations of protocol-based fuzzing based on protocol implementations of published specifications: 1) Testing cannot proceed until the specification is relatively mature, since a specification is a prerequisite for writing such a fuzzer; and 2) Many useful protocols are proprietary, or involve proprietary extensions to published protocols. If fuzzing is based only on published specifications, test coverage for new or proprietary protocols will be limited or nonexistent.

Fuzz testing can be combined with other testing techniques. White-box fuzzing uses symbolic execution and constraint solving.^[15] Evolutionary fuzzing leverages feedback from code coverage,^[16] effectively automating the approach of *exploratory testing*.

Reproduction and isolation

Test case reduction is the process of extracting minimal test cases from an initial test case.^{[17] [18]} Test case reduction may be done manually, or using software tools, and usually involves a divide-and-conquer strategy where parts of the test are removed one by one until only the essential core of the test case remains.

So as to be able to reproduce errors, fuzzing software will often record the input data it produces, usually before applying it to the software. If the computer crashes outright, the test data is preserved. If the fuzz stream is pseudo-random number-generated, the seed value can be stored to reproduce the fuzz attempt. Once a bug is found, some fuzzing software will help to build a test case, which is used for debugging, using test case reduction tools such as Delta or Lithium.

Advantages and disadvantages

The main problem with fuzzing to find program faults is that it generally only finds very simple faults. The computational complexity of the software testing problem is of exponential order ($O(c^n)$, $c > 1$) and every fuzzer takes shortcuts to find something interesting in a timeframe that a human cares about. A primitive fuzzer may have poor code coverage; for example, if the input includes a checksum which is not properly updated to match other random changes, only the checksum validation code will be verified. Code coverage tools are often used to estimate how "well" a fuzzer works, but these are only guidelines to fuzzer quality. Every fuzzer can be expected to find a different set of bugs.

On the other hand, bugs found using fuzz testing are sometimes severe, exploitable bugs that could be used by a real attacker. This has become more common as fuzz testing has become more widely known, as the same techniques and tools are now used by attackers to exploit deployed software. This is a major advantage over binary or source auditing, or even fuzzing's close cousin, fault injection, which often relies on artificial fault conditions that are difficult or impossible to exploit.

The randomness of inputs used in fuzzing is often seen as a disadvantage, as catching a boundary value condition with random inputs is highly unlikely.

Fuzz testing enhances software security and software safety because it often finds odd oversights and defects which human testers would fail to find, and even careful human test designers would fail to create tests for.

References

- [1] Michael Sutton, Adam Greene, Pedram Amini (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley. ISBN 0321446119.
- [2] John Neystadt (2008-02). "Automated Penetration Testing with White-Box Fuzzing" (<http://msdn.microsoft.com/en-us/library/cc162782.aspx>). Microsoft. . Retrieved 2009-05-14.
- [3] Barton Miller (2008). "Preface". In Ari Takanan, Jared DeMott and Charlie Miller, *Fuzzing for Software Security Testing and Quality Assurance*, ISBN 978-1-59693-214-2
- [4] "Fuzz Testing of Application Reliability" (<http://pages.cs.wisc.edu/~bart/fuzz/>). University of Wisconsin-Madison. . Retrieved 2009-05-14.
- [5] "Macintosh Stories: Monkey Lives" (http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt). Folklore.org. 1999-02-22. . Retrieved 2010-05-28.
- [6] Jesse Ruderman. "Fuzzing for correctness" (<http://www.squarefree.com/2007/08/02/fuzzing-for-correctness/>). .
- [7] Jesse Ruderman. "Fuzzing TraceMonkey" (<http://www.squarefree.com/2008/12/23/fuzzing-tracemonkey/>). .
- [8] Jesse Ruderman. "Some differences between JavaScript engines" (<http://www.squarefree.com/2008/12/23/differences/>). .
- [9] "Robustness Testing Of Industrial Control Systems With Achilles" (http://wurldtech.com/resources/SB_002_Robustness_Testing_With_Achilles.pdf) (PDF). . Retrieved 2010-05-28.
- [10] "Software Testing Techniques by Boris Beizer. International Thomson Computer Press; 2 Sub edition (June 1990)" (<http://www.amazon.com/dp/1850328803>). Amazon.com. . Retrieved 2010-05-28.
- [11] "Kaksonen, Rauli. (2001) A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications 447. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.)." (<http://www.vtt.fi/inf/pdf/publications/2001/P448.pdf>) (PDF). . Retrieved 2010-05-28.

- [12] "Software Fault Injection: Inoculating Programs Against Errors by Jeffrey M. Voas and Gary McGraw" (<http://www.amazon.com/dp/0471183814>). John Wiley & Sons. January 28, 1998. .
- [13] <http://sequitur.info/>
- [14] Dan Kaminski (2006). "Black Ops 2006" (<http://usenix.org/events/lisa06/tech/slides/kaminsky.pdf>). .
- [15] Patrice Godefroid, Adam Kiezun, Michael Y. Levin. "Grammar-based Whitebox Fuzzing" (<http://people.csail.mit.edu/akiezun/pldi-kiezun.pdf>). Microsoft Research. .
- [16] "VDA Labs" (http://www.vdalabs.com/tools/efs_gpf.html). .
- [17] "Test Case Reduction" (<http://www.webkit.org/quality/reduction.html>). 2011-07-18. .
- [18] "IBM Test Case Reduction Techniques" (<https://www-304.ibm.com/support/docview.wss?uid=swg21084174>). 2011-07-18. .

Further reading

- ISBN 978-1-59693-214-2, *Fuzzing for Software Security Testing and Quality Assurance*, Ari Takanen, Jared D. DeMott, Charles Miller

External links

- University of Wisconsin Fuzz Testing (the original fuzz project) (<http://www.cs.wisc.edu/~bart/fuzz>) Source of papers and fuzz software.
- Look out! It's the Fuzz! (IATAC IAnewsletter 10-1) (http://iac.dtic.mil/iatac/download/Vol10_No1.pdf)
- Designing Inputs That Make Software Fail (<http://video.google.com/videoplay?docid=6509883355867972121>), conference video including fuzzy testing
- Link to the Oulu (Finland) University Secure Programming Group (<http://www.ee.oulu.fi/research/ouspg/>)
- JBroFuzz - Building A Java Fuzzer (<http://video.google.com/videoplay?docid=-1551704659206071145>), conference presentation video
- Building 'Protocol Aware' Fuzzing Frameworks (<http://docs.google.com/viewer?url=https://github.com/s7ephen/Ruxxer/raw/master/presentations/Ruxxer.ppt>)

Game testing

Game testing, a subset of game development, is a software testing process for quality control of video games.^{[1] [2]}^[3] The primary function of game testing is the discovery and documentation of software defects (aka bugs). Interactive entertainment software testing is a highly technical field requiring computing expertise, analytic competence, critical evaluation skills, and endurance.^{[4] [5]}

History

In the early days of computer and video games, the developer was in charge of all the testing. No more than one or two testers were required due to the limited scope of the games. In some cases, the programmers could handle all the testing.

As games become more complex, a larger pool of QA resources, called "Quality Assessment" or "Quality Assurance" is necessary. Most publishers employ a large QA staff for testing various games from different developers. Despite the large QA infrastructure most publishers have, many developers retain a small group of testers to provide on-the-spot QA.

A common misconception is that all game testers enjoy alpha or beta version of the game and report occasionally found bugs.^[5] In contrast, game testing is highly focused on finding bugs using established and often tedious methodologies before alpha version.

Overview

Quality assurance is a critical component in game development, though the video game industry does not have a standard methodology. Instead developers and publishers have their own methods. Small developers do not have QA staff, however large companies may employ QA teams full-time. High-profile commercial games are professionally and efficiently tested by publisher QA department.^[6]

Testing starts as soon as first code is written and increases as the game progresses towards completion.^{[7] [8]} The main QA team will monitor the game from its first submission to the QA until as late as post-production.^[8] Early in the game development process the testing team is small and focuses on daily feedback for new code. As the game approaches *alpha* stage, more team members are employed and test plan is written. Sometimes features that are not bugs are reported as bugs and sometimes programming team fails to fix issues first time around.^[9] A good bug-reporting system may help the programmers work efficiently. As the projects enters *beta* stage, the testing team will have clear assignments for each day. Tester feedback may determine final decisions of exclusion or inclusion of final features. Introducing previously uninvolved testers with fresh perspective may help identify new bugs.^{[8] [10]} At this point the lead tester communicates with the producer and department heads daily.^[11] If the developer has external publisher, then coordination with publisher's QA team starts. For console games, a build for console company QA team is sent. Beta testing may involve volunteers, for example, if the game is multiplayer.^[10]

Testers receive scheduled uniquely identifiable game builds^[10] from the developers. The game is play-tested and testers note any uncovered errors. These may range from bugs to art glitches to logic errors and level bugs. Testing requires creative gameplay to discover often subtle bugs. Some bugs are easy to document, but many require detailed description so a developer can replicate or find the bug. Testers implement concurrency control to avoid logging bugs multiple times. Many video game companies separate technical requirement testing from functionality testing altogether since a different testing skillset is required.^[5]

If a video game development enters crunch time before a deadline, the game-test team is required to test late-added features and content without delay. During this period staff from other departments may contribute to the testing—especially in multiplayer games.

Most companies rank bugs according to an estimate of their severity:^[12]

- *A bugs* are critical bugs that prevent the game from being shipped, for example, they may crash the game.^[10]
- *B bugs* are essential problems that require attention, however the game may still be playable. Multiple B bugs are equally severe to an A bug.^[10]
- *C bugs* are small and obscure problems, often in form of recommendation rather than bugs.^[11]

Game tester

A game tester is a member of a development team who performs game testing.

Roles

The organization of staff differs between organizations; a typical company may employ the following roles associated with testing disciplines:

- *Game producers* are responsible for setting testing deadlines in coordination with marketing and quality assurance.^[13] They also manage many items outside of game testing, relating to the overall production of a title. Their approval is typically required for final submission or "gold" status.
- *Lead tester, test lead*^[9] or *QA lead*^[6] is the person responsible for the game working correctly^[9] and managing bug lists.^[10] A lead tester manages the QA staff.^[6] Lead tester works closely with designers and programmers, especially towards the end of the project. Lead tester is responsible for tracking bug reports and managing that they are fixed.^[9] They are also responsible that QA teams produce formal and complete reports.^[10] This includes discarding duplicate and erroneous bug reports, as well as requesting clarifications.^[6] As the game nears alpha and beta stages, lead tester brings more testers into the team, coordinates with external testing teams and works with management and producers.^[12] Some companies may prevent the game going gold until lead tester approves it.^[11] Lead testers are also typically responsible for compiling representative samples of game footage for submission to regulatory bodies such as the ESRB and PEGI. A lead tester is often an aspiring designer or producer.^[6]
- *Testers* are responsible for checking that the game works, is easy to use, has actions that make sense, and contains fun gameplay.^[11] Testers need to write accurate and specific bug reports, and if possible providing descriptions of how the bug can be reproduced.^[14] Testers may be assigned to a single game during its entire production, or brought onto other projects as demanded by the department's schedule and specific needs.

Employment

Game QA is less technical than general software QA. Many game testers require little experience and sometimes only a high school diploma and no technical expertise will suffice.^{[6] [15]} Game testing is normally a full-time job for experienced testers,^[16] however many employees are hired as temporary staff,^{[2] [17]} such as beta testers. In some cases, testers employed by a publisher may be sent to work at the developer's site. The most aggressive recruiting season is late summer/early autumn, as this is the start of the crunch period for games to be finished and shipped in time for the holiday season.

Some testers use the job as a stepping stone in the game industry.^{[15] [3]} QA résumés, which display non-technical skill sets, tend towards management, then to marketing or production. Applicants for programming, art, or design positions need to demonstrate technical skills in these areas.^[18]

Compensation

Game testing personnel are usually paid hourly (around US\$10–12 an hour). Testing management is usually more lucrative, and requires experience and often a college education. An annual survey found that testers earn an average of \$39k annually. Testers with less than three years experience earn an average of US\$25k while testers with over three years experience earn US\$43k. Testing leads, with over six years experience, earn on an average of US\$71k a year. ^[19] Typically, they will make \$35-45k with less experience.

Some employers offer bonuses for the number of bugs found.

Process

A typical bug report progression of testing process is seen below:

- *Identification.* Incorrect program behaviour is analyzed and identified as a bug.
- *Reporting.* The bug is reported to the developers using a defect tracking system. The circumstances of the bug and steps to reproduce are included in the report. Developers may request additional documentation such as a real-time video of the bug's manifestation.
- *Analysis.* The developer responsible for the bug, such as an artist, programmer or game designer checks the malfunction. This is outside the scope of game tester duties, although inconsistencies in the report may require more information or evidence from the tester.
- *Verification.* After the developer fixes the issue, the tester verifies that the bug no longer occurs. Not all bugs are addressed by the developer, for example, some bugs may be claimed as features (expressed as "NAB" or "not a bug"), and may also be "waived" (given permission to be ignored) by producers, game designers, or even lead testers, according to company policy.

Methodology

There is no industry standard method for game testing, and most methodologies are developed by individual video game developers and publishers. Methodologies are continuously refined and may differ for different types of games (for example, the methodology for testing a MMORPG will be different from testing a casual game). Many methods, such as unit testing, are borrowed directly from general software testing techniques. Outlined below are the most important methodologies, specific to video games.

- **Functionality testing** is most commonly associated with the phrase "game testing", as it entails playing the game in some form. Functionality testing does not require extensive technical knowledge. Functionality testers look for general problems within the game itself or its user interface, such as stability issues, game mechanic issues, and game asset integrity.
- **Compliance testing** is the reason for the existence of game testing labs. First-party licensors for console platforms have strict technical requirements titles licensed for their platforms. For example, Sony publishes a *Technical Requirements Checklist* (TRC), Microsoft publishes *Technical Certification Requirements* (TCR), and Nintendo publishes a set of "guidelines" (Lotcheck). Some of these requirements are highly technical and fall outside the scope of game testing. Other parts, most notably the formatting of standard error messages, handling of memory card data, and handling of legally trademarked and copyrighted material, are the responsibility of the game testers. Even a single violation in submission for license approval may have the game rejected, possibly incurring additional costs in further testing and resubmission. In addition, the delay may cause the title to miss an important launch window, potentially costing the publisher even larger sums of money.

The requirements are proprietary documents released to developers and publishers under confidentiality agreements. They are not available for the general public to review, although familiarity with these standards is considered a valuable skill to have as a tester.

Compliance may also refer to regulatory bodies such as the ESRB and PEGI, if the game targets a particular content rating. Testers must report objectionable content that may be inappropriate for the desired rating. Similar to licensing, games that do not receive the desired rating must be re-edited, retested, and resubmitted at additional cost.

- **Compatibility testing** is normally required for PC titles, nearing the end of development as much of the compatibility depends on the final build of the game. Often two rounds of compatibility tests are done - early in beta to allow time for issue resolution, and late in beta or during release candidate. Compatibility testing team test major functionality of the game on various configurations of hardware. Usually a list of commercially important hardware is supplied by the publisher.^[8]

Compatibility testing ensures that the game runs on different configurations of hardware and software. The hardware encompasses brands of different manufacturers and assorted input peripherals such as gamepads and joysticks.

The testers also evaluate performance and results are used for game's advertised minimum system requirements. Compatibility or performance issues may be either fixed by the developer or, in case of legacy hardware and software, support may be dropped.

- **Localization testing** act as in-game text editors.^[2] Although general text issues are a part of functionality testing, QA departments may employ dedicated localization testers. In particular, early Japanese game translations were rife with English, and in recent years localization testers are employed to make technical corrections and review translation work of game scripts^[20] - catalogued collections of all the in-game text. Testers native to the region where a game is marketed may be employed to ensure the accuracy and quality of a game's localization.^[8]
- **Soak testing**, in the context of video games, involves leaving the game running for prolonged periods time in various modes of operation, such as idling, paused, or at the title screen. This testing requires no user interaction beyond initial setup, and is usually managed by lead testers. Automated tools may be used for simulating repetitive actions, such mouse clicks. Soaking can detect memory leaks or rounding errors that manifest only over time. Soak tests are one of the compliance requirements.
- **Beta testing** is done during beta stage of development. Often this refers to the first publicly available version of a game. Public betas are effective because thousands of fans may find bugs that the developer's testers did not.
- **Regression testing** is performed once a bug has been fixed by the programmers. QA checks to see whether the bug is still there (regression) and then runs similar tests to see whether the fix broke something else. That second stage is often called "halo testing"; it involves testing all around a bug, looking for other bugs.
- **Load testing** tests the limits of a system, such as the number of players on an MMO server, the number of sprites active on the screen, or the number of threads running in a particular program. Load testing requires either a large group of testers or software that emulates heavy activity.^[2]
- **Multiplayer testing** may involve separate multiplayer QA team if the game has significant multiplayer portions. This testing is more common with PC games. The testers ensure that all connectivity methods (modem, LAN, Internet) are working. This allows single player and multiplayer testing to occur in parallel.^[8]

Console hardware

For consoles, the majority of testing is not performed on a normal system or *consumer unit*. Special test equipment is provided to developers and publishers. The most significant tools are the *test* or *debug* kits, and the *dev* kits. The main difference between from consumer units is the ability to load games from a burned disc or from a hard drive, as well as being able to set the console for any publishing region. This allows game developers to produce copies for testing. This functionality is not present in consumer units to combat software piracy and grey-market imports.

- *Test kits* have the same hardware specifications and overall appearance as a consumer unit, though often with additional ports and connectors for other testing equipment. Test kits contain additional options, such as running

automated compliance checks, especially with regard to save data. The system software also allows the user to capture memory dumps for aid in debugging.

- *Dev kits* are not normally used by game testers, but are used by programmers for lower-level testing. In addition to the features of a test kit, dev kits usually have higher hardware specifications, most notably increased system memory. This allows developers to estimate early game performance without worrying about optimizations. Dev kits are usually larger and look different from a test kit or consumer unit.

Notes

- [1] Bates 2004, pp. 176-180
- [2] Moore, Novak 2010, p. 95
- [3] Oxland 2004, p. 301-302
- [4] Bates 2004, pp. 178, 180
- [5] Oxland 2004, p. 301
- [6] Bethke 2003, p. 52
- [7] Bates 2004, p. 176
- [8] Bethke 2003, p. 53
- [9] Bates 2004, p. 177
- [10] Bates 2004, p. 178
- [11] Bates 2004, p. 179
- [12] Bates 2004, pp. 178-179
- [13] Moore, Novak 2010, p. 72
- [14] Bates 2004, p. 180
- [15] Bates 2004, p. 261
- [16] Moore, Novak 2010, p. 25
- [17] Moore, Novak 2010, p. 2
- [18] Moore, Novak 2010, pp. 84, 237-238
- [19] Fleming, Jeffrey (2008). "7th Annual Salary Survey". *Game Developer* (United Business Media) **15** (4): 8.
- [20] Adams, Rollings 2003, p. 17

References

- Adams, Ernest; Rollings, Andrew (2003). *Andrew Rollings and Ernest Adams on game design*. New Riders Publishing. ISBN 1592730019.
- Bates, Bob (2004). *Game Design* (2nd ed.). Thomson Course Technology. ISBN 1592004938.
- Moore, Michael E.; Novak, Jeannie (2010). *Game Industry Career Guide*. Delmar: Cengage Learning. ISBN 978-1-4283-7647-2.
- Oxland, Kevin (2004). *Gameplay and design*. Addison Wesley. ISBN 0321204670.

External links

- Sloperama: "Working as a Tester" (<http://www.sloperama.com/advice/lesson5.htm>) from game industry veteran, Tom Sloper
- Game tester (http://www.igda.org/breakingin/path_production.htm#test) from the International Game Developers Association (IGDA)
 - IGDA's Quality Assurance Special Interest Group (<http://www.igda.org/qa>)
- Book Extract: Game QA & Testing - Ready, Set, Go! (http://www.gamecareerguide.com/features/764/book_extract_game_qa_testing_.php)
- "Testing Video Games Can't Possibly Be Harder Than an Afternoon With Xbox, Right?" (<http://www.seattleweekly.com/2007-07-11/news/testing-video-games-can-t-possibly-be-harder-than-an-afternoon-with-xbox-right.php?page=full>) from *Seattle Weekly*

- "Game Tester Interview" (<http://www.gamertagrado.com/forums/showthread.php?t=6435>) from *Gamer Tag Radio*

Google Guice

Google Guice

Developer(s)	Google
Stable release	2.0.0 / May 19, 2009
Preview release	3.0rc3 / March 6, 2011
Development status	Active
Written in	Java
Operating system	Cross-platform
Type	dependency injection framework
License	Apache License 2.0
Website	[1]

Google Guice is an open source software framework for the Java platform released by Google under an Apache license. It provides support for dependency injection using annotations to configure Java objects.^[2] Dependency injection is a design pattern whose core principle is to separate behavior from dependency resolution.

Guice allows implementation classes to be programmatically bound to an interface, then injected into constructors, methods or fields using an `@Inject` annotation. When more than one implementation of the same interface is needed, the user can create custom annotations that identify an implementation, then use that annotation when injecting it.

Being the first generic framework for dependency injection using java annotations in 2008, Guice won the 18th Jolt Award for best Library, Framework, or Component^[2] ^[3].

References

- [1] <http://code.google.com/p/google-guice/>
- [2] Guice (Google) (<http://www.drdoobs.com/mobility/207600666;jsessionid=WZRTJLDOCG00RQE1GHOSKH4ATMY32JVN?pgno=9>), Reviewed by Michael Yuan, retrieved 2010-04-09.
- [3] 18th Annual Jolt Award winners (<http://www.joltawards.com/history/winners.html>)

Further reading

- Vanbrabant, Robbie (April 21, 2008), *Google Guice: Agile Lightweight Dependency Injection Framework* (<http://www.apress.com/book/view/1590599977>) (1st ed.), Apress, pp. 192, ISBN 978-1590599976

External links

- [google-guice - Google Code \(http://code.google.com/p/google-guice/\)](http://code.google.com/p/google-guice/)
- [Extensions for Google Guice \(http://code.google.com/docreader/#p=google-guice&s=google-guice&t=Extensions\)](http://code.google.com/docreader/#p=google-guice&s=google-guice&t=Extensions)
- [Warp Extensions for Google Guice \(http://www.wideplay.com\)](http://www.wideplay.com)
- [Big Modular Java with Guice \(http://www.java-tv.com/2009/09/28/big-modular-java-with-guice/\)](http://www.java-tv.com/2009/09/28/big-modular-java-with-guice/)
- [Guice and @Inject - Stuart McCulloch \(http://vimeo.com/17156850\)](http://vimeo.com/17156850)

Graphical user interface testing

In software engineering, **graphical user interface testing** is the process of testing a product's graphical user interface to ensure it meets its written specifications. This is normally done through the use of a variety of test cases.

Test Case Generation

To generate a 'good' set of test cases, the test designers must be certain that their suite covers all the functionality of the system and also has to be sure that the suite fully exercises the GUI itself. The difficulty in accomplishing this task is twofold: one has to deal with domain size and then one has to deal with sequences. In addition, the tester faces more difficulty when they have to do regression testing.

The size problem can be easily illustrated. Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A relatively small program such as Microsoft WordPad has 325 possible GUI operations.^[1] In a large program, the number of operations can easily be an order of magnitude larger.

The second problem is the sequencing problem. Some functionality of the system may only be accomplished by following some complex sequence of GUI events. For example, to open a file a user may have to click on the File Menu and then select the Open operation, and then use a dialog box to specify the file name, and then focus the application on the newly opened window. Obviously, increasing the number of possible operations increases the sequencing problem exponentially. This can become a serious issue when the tester is creating test cases manually.

Regression testing becomes a problem with GUIs as well. This is because the GUI may change significantly across versions of the application, even though the underlying application may not. A test designed to follow a certain path through the GUI may not be able to follow that path since a button, menu item, or dialog may have changed location or appearance.

These issues have driven the GUI testing problem domain towards automation. Many different techniques have been proposed to automatically generate test suites that are complete and that simulate user behavior.

Most of the techniques used to test GUIs attempt to build on techniques previously used to test CLI (Command Line Interface) programs. However, most of these have scaling problems when they are applied to GUI's. For example, Finite State Machine-based modeling^{[2] [3]} — where a system is modeled as a finite state machine and a program is used to generate test cases that exercise all states — can work well on a system that has a limited number of states but may become overly complex and unwieldy for a GUI (see also model-based testing).

Planning and artificial intelligence

A novel approach to test suite generation, adapted from a **CLI** technique^[4] involves using a planning system.^[5] Planning is a well-studied technique from the artificial intelligence (AI) domain that attempts to solve problems that involve four parameters:

- an initial state,
- a goal state,
- a set of operators, and
- a set of objects to operate on.

Planning systems determine a path from the initial state to the goal state by using the operators. An extremely simple planning problem would be one where you had two words and one operation called 'change a letter' that allowed you to change one letter in a word to another letter – the goal of the problem would be to change one word into another.

For GUI testing, the problem is a bit more complex. In ^[1] the authors used a planner called IPP^[6] to demonstrate this technique. The method used is very simple to understand. First, the systems UI is analyzed to determine what operations are possible. These operations become the operators used in the planning problem. Next an initial system state is determined. Next a goal state is determined that the tester feels would allow exercising of the system. Lastly the planning system is used to determine a path from the initial state to the goal state. This path becomes the test plan.

Using a planner to generate the test cases has some specific advantages over manual generation. A planning system, by its very nature, generates solutions to planning problems in a way that is very beneficial to the tester:

1. The plans are always valid. What this means is that the output of the system can be one of two things, a valid and correct plan that uses the operators to attain the goal state or no plan at all. This is beneficial because much time can be wasted when manually creating a test suite due to invalid test cases that the tester thought would work but didn't.
2. A planning system pays attention to order. Often to test a certain function, the test case must be complex and follow a path through the GUI where the operations are performed in a specific order. When done manually, this can lead to errors and also can be quite difficult and time consuming to do.
3. Finally, and most importantly, a planning system is goal oriented. What this means and what makes this fact so important is that the tester is focusing test suite generation on what is most important, testing the functionality of the system.

When manually creating a test suite, the tester is more focused on how to test a function (i. e. the specific path through the GUI). By using a planning system, the path is taken care of and the tester can focus on what function to test. An additional benefit of this is that a planning system is not restricted in any way when generating the path and may often find a path that was never anticipated by the tester. This problem is a very important one to combat.^[7]

Another interesting method of generating GUI test cases uses the theory that good GUI test coverage can be attained by simulating a novice user. One can speculate that an expert user of a system will follow a very direct and predictable path through a GUI and a novice user would follow a more random path. The theory therefore is that if we used an expert to test the GUI, many possible system states would never be achieved. A novice user, however, would follow a much more varied, meandering and unexpected path to achieve the same goal so it's therefore more desirable to create test suites that simulate novice usage because they will test more.

The difficulty lies in generating test suites that simulate 'novice' system usage. Using Genetic algorithms is one proposed way to solve this problem.^[7] Novice paths through the system are not random paths. First, a novice user will learn over time and generally won't make the same mistakes repeatedly, and, secondly, a novice user is not analogous to a group of monkeys trying to type Hamlet, but someone who is following a plan and probably has some domain or system knowledge.

Genetic algorithms work as follows: a set of 'genes' are created randomly and then are subjected to some task. The genes that complete the task best are kept and the ones that don't are discarded. The process is again repeated with the surviving genes being replicated and the rest of the set filled in with more random genes. Eventually one gene (or a small set of genes if there is some threshold set) will be the only gene in the set and is naturally the best fit for the given problem.

For the purposes of the GUI testing, the method works as follows. Each gene is essentially a list of random integer values of some fixed length. Each of these genes represents a path through the GUI. For example, for a given tree of widgets, the first value in the gene (each value is called an allele) would select the widget to operate on, the following alleles would then fill in input to the widget depending on the number of possible inputs to the widget (for example a pull down list box would have one input...the selected list value). The success of the genes are scored by a criterion that rewards the best 'novice' behavior.

The system to do this testing described in^[7] can be extended to any windowing system but is described on the X window system. The X Window system provides functionality (via XServer and the editors' protocol) to dynamically send GUI input to and get GUI output from the program without directly using the GUI. For example, one can call XSendEvent() to simulate a click on a pull-down menu, and so forth. This system allows researchers to automate the gene creation and testing so for any given application under test, a set of novice user test cases can be created.

Running the test cases

At first the strategies were migrated and adapted from the CLI testing strategies. A popular method used in the CLI environment is capture/playback. Capture playback is a system where the system screen is "captured" as a bitmapped graphic at various times during system testing. This capturing allowed the tester to "play back" the testing process and compare the screens at the output phase of the test with expected screens. This validation could be automated since the screens would be identical if the case passed and different if the case failed.

Using capture/playback worked quite well in the CLI world but there are significant problems when one tries to implement it on a GUI-based system.^[8] The most obvious problem one finds is that the screen in a GUI system may look different while the state of the underlying system is the same, making automated validation extremely difficult. This is because a GUI allows graphical objects to vary in appearance and placement on the screen. Fonts may be different, window colors or sizes may vary but the system output is basically the same. This would be obvious to a user, but not obvious to an automated validation system.

To combat this and other problems, testers have gone 'under the hood' and collected GUI interaction data from the underlying windowing system.^[9] By capturing the window 'events' into logs the interactions with the system are now in a format that is decoupled from the appearance of the GUI. Now, only the event streams are captured. There is some filtering of the event streams necessary since the streams of events are usually very detailed and most events aren't directly relevant to the problem. This approach can be made easier by using an MVC architecture for example and making the view (i. e. the GUI here) as simple as possible while the model and the controller hold all the logic. Another approach is to use the software's built-in assistive technology, to use an HTML interface or a three-tier architecture that makes it also possible to better separate the user interface from the rest of the application.

Another way to run tests on a GUI is to build a driver into the GUI so that commands or events can be sent to the software from another program.^[7] This method of directly sending events to and receiving events from a system is highly desirable when testing, since the input and output testing can be fully automated and user error is eliminated.

References

- [1] Atif M. Memon, M.E. Pollack and M.L. Soffa. Using a Goal-driven Approach to Generate Test Cases for GUIs.
- [2] J.M. Clarke. Automated test generation from a Behavioral Model. In Proceedings of Pacific Northwest Software Quality Conference. IEEE Press, May 1998.
- [3] S. Esmelioglu and L. Apfelbaum. Automated Test generation, execution and reporting. In Proceedings of Pacific Northwest Software Quality Conference. IEEE Press, October 1997.
- [4] A. Howe, A. von Mayrhauser and R.T. Mraz. Test case generation as an AI planning problem. *Automated Software Engineering*, 4:77-106, 1997.
- [5] "Hierarchical GUI Test Case Generation Using Automated Planning" by Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, 2001, pp. 144-155, IEEE Press.
- [6] J. Koehler, B. Nebel, J. Hoffman and Y. Dimopoulos. Extending planning graphs to an ADL subset. *Lecture Notes in Computer Science*, 1348:273, 1997.
- [7] D.J. Kasik and H.G. George. Toward automatic generation of novice user test scripts. In M.J. Tauber, V. Bellotti, R. Jeffries, J.D. Mackinlay, and J. Nielsen, editors, *Proceedings of the Conference on Human Factors in Computing Systems : Common Ground*, pages 244-251, New York, 13–18 April 1996, ACM Press. (http://www.sigchi.org/chi96/proceedings/papers/Kasik/djk_txt.htm)
- [8] L.R. Kepple. The black art of GUI testing. *Dr. Dobb's Journal of Software Tools*, 19(2):40, Feb. 1994.
- [9] M.L. Hammontree, J.J. Hendrickson and B.W. Hensley. Integrated data capture and analysis tools for research and testing on graphical user interfaces. In P. Bauersfeld, J. Bennett and G. Lynch, editors, *Proceedings of the Conference on Human Factors in Computing System*, pages 431-432, New York, NY, USA, May 1992. ACM Press.

External links

- Article GUI Testing Checklist (<http://www.methodsandtools.com/archive/archive.php?id=37>)
- GUITAR GUI Testing Software (<http://guitar.sourceforge.net/>)
- Event-Driven Software Lab (<http://www.cs.umd.edu/~atif/edsl>)
- NUnitForms (<http://nunitforms.sourceforge.net/>) an add-on to the popular testing framework NUnit for automatic GUI testing of WinForms applications
- GUI Test Drivers (<http://www.testingfaqs.org/t-gui.html>) Lists and describes tools resp. frameworks in different programming languages
- <http://www.youtube.com/watch?v=6LdsIVvxISU> (<http://www.youtube.com/watch?v=6LdsIVvxISU>) A talk at the Google Test Automation Conference by Prof. Atif M Memon (<http://www.cs.umd.edu/~atif>) on Model-Based GUI Testing.
- Testing GUI Applications (<http://www.gerrardconsulting.com/?q=node/514>) A talk at EuroSTAR 97, Edinburgh UK by Paul Gerrard.
- Xnee, a program that can be used to record and replay test.

Hybrid testing

Overview

The hybrid Test Automation Framework is what most frameworks evolve into over time and multiple projects. The most successful automation frameworks generally accommodate both Keyword-driven testing as well as Data-driven testing. This allows data driven scripts to take advantage of the powerful libraries and utilities that usually accompany a keyword driven architecture. The framework utilities can make the data driven scripts more compact and less prone to failure than they otherwise would have been. The utilities can also facilitate the gradual and manageable conversion of existing scripts to keyword driven equivalents when and where that appears desirable. On the other hand, the framework can use scripts to perform some tasks that might be too difficult to re-implement in a pure keyword driven approach, or where the keyword driven capabilities are not yet in place.

The Framework

The framework is defined by the Core Data Driven Engine, the Component Functions, and the Support Libraries (see adjacent picture) . While the Support Libraries provide generic routines useful even outside the context of a keyword driven framework, the core engine and component functions are highly dependent on the existence of all three elements. The test execution starts with the LAUNCH TEST(1) script. This script invokes the Core Data Driven Engine by providing one or more High-Level Test Tables to CycleDriver(2). CycleDriver processes these test tables invoking the SuiteDriver(3) for each Intermediate-Level Test Table it encounters. SuiteDriver processes these intermediate-level tables invoking StepDriver(4) for each Low-Level Test Table it encounters. As StepDriver processes these low-level tables it attempts to keep the application in synch with the test. When StepDriver encounters a low-level command for a specific component, it determines what Type of component is involved and invokes the corresponding Component Function(5) module to handle the task.

IBM Product Test

IBM Product Test was a group level organization for testing of IBM hardware and software products as part of the product *development* rather than *manufacturing*. It was housed in a number of laboratories associated with the various manufacturing and software development facilities. As an independent organization, Group Product Test had much power and exercised strict control over product quality. Ultimately, however, its contribution to the bottom line could not be quantified, and it was disbanded in 197?.

Types of testing

Two types of test were performed:

1. A- or Alpha-test: These were design and feasibility tests, carried out prior to announcement. In some cases, software modelling was used to estimate performance.
2. B- or Beta-test: These tests were carried out to support a First Customer Ship (FCS) date for software, or a commit to manufacturing for hardware.

Other functions

Keeper of Standards

Product Test was responsible for keeping and maintaining the body of standards governing the various aspects of design and development of IBM products. Proposed standards were circulated by the Standards Coordinator to other Product Test Laboratories for approval.

Special tests

Tape drive reel hub: When IBM introduced a new form of tape drive reel hub using a latch mechanism to replace the older screw-on hub, failure reports began to come in from the field. Improvements were made to the design and subjected to extensive testing with periodic inspections. Initially, the testing was performed manually by repeated mounting and dismounting a tape reel, with everyone in the Lab (Poughkeepsie) assigned a number of operations, including the Lab Director. Eventually, a robot was constructed to perform these exhausting tests.

Product Test Laboratories

- Poughkeepsie, NY
 - Endicott, NY
 - LaGaude, (Nice/France
 - Montpellier, France
 - Boeblingen, Germany
 - and others
-

IBM Rational Quality Manager

IBM Rational Quality Manager

Developer(s)	IBM / Rational Software
Stable release	3.0.1 / June 2011
Development status	Active
Operating system	Microsoft Windows Linux
Type	Test management tools
License	Proprietary
Website	Rational Quality Manager Web Page ^[1]

IBM Rational Quality Manager provides a collaborative application lifecycle management environment for test planning, construction, and execution.

Overview

IBM Rational Quality Manager enables quality assurance teams to track all aspects of the quality assurance effort. The central artifact in the tool is a dynamic test plan that contains all information pertaining to the quality assurance effort, such as goals, schedules, milestones and exit criteria as well as links to associated test cases, requirements and development work items. Rational Quality Manager further includes modules for requirements management, manual test authoring and execution, test lab management, test execution, reporting and defect management.

Quality Manager is a web 2.0 application which runs in a browser. Data is stored and managed on the Rational Quality Manager server.

It is built upon the IBM Rational Jazz technology platform. The Jazz technology platform is a common server foundation shared by several Rational tools which facilitates information sharing between teams and applications. Through the Jazz technology platform, Rational Quality Manager can share requirements information with Rational Requirements Composer, and share work items and defects with Rational Team Concert. The Jazz platform also provides a universal API for sharing information with other 3rd party applications.

IBM Rational Quality Manager and the Jazz technology platform are developed on Jazz.net ^[2], where developers and users participate and communicate in discussion forums, newsgroups, and the development process. The Jazz.net community site includes technotes, articles, forums, wikis, blogs, current documentation, and other troubleshooting and support resources.

Rational Quality Manager Key Artifacts

Requirements

Requirements are a key input to the quality assurance process. IBM Rational Quality Manager provides two requirements management options. First, IBM Rational Quality Manager can integrate to external requirements management systems such as Rational DOORS, Rational RequisitePro and Rational Requirements Composer. In this scenario, requirements are managed externally, and Rational Quality Manager establishes live links to the external requirements. Alternatively, Rational Quality Manager contains its own requirements management facility. This streamlined facility enables requirements management direction from the Quality Manager interface. Regardless of the source of requirements, Rational Quality manager can generate test cases from requirements, can associated test

plans and test cases to requirements and will notify the user when a requirement associated to a test plan or test case has changed.

Test Plan

The test plan is the central artifact in Rational Quality Manager and contains both static and dynamic information regarding the quality assurance effort. Some of the static information gathered includes Business Objectives, Test Objectives, Test Team, Entry and Exit Criteria. Some of the dynamic information contained in the test plan includes requirements and test cases.

Test Cases

Test cases are the single most important element of the test plan. The test case defines the what, where, why and how of a test. Specifically, what is to be tested, on which platforms it is to be tested, the purpose of the test and how the test will be executed. In terms of execution, Rational Quality Manager has the native ability to author and execute manual test scripts as well as the ability to launch automated test scripts from Rational Functional Tester, Rational Performance Tester and Rational Service Tester. Various execution adapters are also available which enable execution of other tests from third party vendors.

Test Lab Assets

Test lab management involves the tracking of test lab assets and their configurations, provisioning of test lab assets and scheduling of test lab assets. Rational Quality Manager provides a basic lab management capability as a standard feature and also offers additional functionality from Rational Test Lab Manager for additional lab management functionality, such as integrations to automated lab discovery and provisioning tools from IBM Tivoli.

Defects

Defect reports are one of the key outputs of the quality assurance process. Like requirements management, Rational Quality Manager offers the ability to manage defects natively, within the Rational Quality Manager architecture or to establish links to external issue tracking solutions such as Rational ClearQuest and Rational Team Concert.

Reports

Rational Quality Manager offers a series of packaged reports for reporting on all aspects of the quality assurance process. With the v2.0 release, Rational Quality Manager has begun integrating new advanced reporting functionality which enables users to create and modify additional reports.

Dashboard

The Rational Quality Manager dashboard itself is not a key artifact, but the default interface which displays artifact information. The dashboard in Rational Quality Manager is fully customizable so that users can choose what content they would like to have on their desktops at all times. The dashboard can display information from Rational Quality Manager as well as from RSS feeds.

Release history

The following is a release history of IBM Rational Quality Manager.

- **v1.00** Released October, 2008. Initial Release
- **v1.01** Released March, 2009.
- **v2.00** Released July, 2009.
- **v2.0.0.1** Released October, 2009.
- **v2.0.1** Released March, 2010
- **v3.0.1** Released June, 2011

Criticisms

Having been in the market for less than 24 months, Rational Quality Manager's primary criticism relates to its newness. This is an issue for all new products, and one that should dissolve with time.

The Rational Quality Manager provides an API which accepts REST requests to create, read, update and delete artifacts and attachments, but there is no reference manual that describes the functionality.

The standard reports in the reporting section are not very customizable and have a plain appeal, e.g. there's no standard report to list requirements, test cases and execution result on the same sheet.

References

[1] <http://jazz.net/projects/rational-quality-manager/>

[2] <http://jazz.net/>

- Rational Quality Manager Web Page (<http://www.ibm.com/software/awdtools/rqm/standard/>)
- Rational Quality Manager Product Documentation (http://www.ibm.com/support/docview.wss?rs=3580&context=SSMKW6&context=SSR27Q&context=SSMKVL&context=SSSTY3&uid=swg27014016&loc=en_US&cs=UTF-8&lang=en)
- Rational Quality Manager Technical Support (<http://www.ibm.com/software/awdtools/rqm/standard/support/doc.html>)
- Rational Quality Manager Download (<http://www.ibm.com/developerworks/downloads/r/rqm/learn.html>)
- Jazz Community Site for Rational Quality Manager (<https://jazz.net/projects/rational-quality-manager/>)
- Rational Quality Manager Blog (<http://qualitymanager.wordpress.com>)

IEEE 829

IEEE Software Document Definitions
SQAP – Software Quality Assurance Plan IEEE 730
SCMP – Software Configuration Management Plan IEEE 828
STD – Software Test Documentation IEEE 829
SRS – Software Requirements Specification IEEE 830
SVVP – Software Validation & Verification Plan IEEE 1012
SDD – Software Design Description IEEE 1016
SPMP – Software Project Management Plan IEEE 1058

IEEE 829-1998, also known as the **829 Standard for Software Test Documentation**, is an IEEE standard that specifies the form of a set of documents for use in eight defined stages of software testing, each stage potentially producing its own separate type of document. The standard specifies the format of these documents but does not stipulate whether they all must be produced, nor does it include any criteria regarding adequate content for these documents. These are a matter of judgment outside the purview of the standard. The documents are:

- **Test Plan: a management planning document** that shows:
 - How the testing will be done (including SUT (system under test) configurations).
 - Who will do it
 - What will be tested
 - How long it will take (although this may vary, depending upon resource availability).
 - What the test coverage will be, i.e. what quality level is required
 - **Test Design Specification:** detailing test conditions and the expected results as well as test pass criteria.
 - **Test Case Specification:** specifying the test data for use in running the test conditions identified in the Test Design Specification
 - **Test Procedure Specification:** detailing how to run each test, including any set-up preconditions and the steps that need to be followed
 - **Test Item Transmittal Report:** reporting on when tested software components have progressed from one stage of testing to the next
 - **Test Log:** recording which tests cases were run, who ran them, in what order, and whether each test passed or failed
 - **Test Incident Report:** detailing, for any test that failed, the actual versus expected result, and other information intended to throw light on why a test has failed. This document is deliberately named as an incident report, and not a fault report. The reason is that a discrepancy between expected and actual results can occur for a number of reasons other than a fault in the system. These include the expected results being wrong, the test being run wrongly, or inconsistency in the requirements meaning that more than one interpretation could be made. The report consists of all details of the incident such as actual and expected results, when it failed, and any supporting evidence that will help in its resolution. The report will also include, if possible, an assessment of the impact of an incident upon testing.
 - **Test Summary Report:** A management report providing any important information uncovered by the tests accomplished, and including assessments of the quality of the testing effort, the quality of the software system under test, and statistics derived from Incident Reports. The report also records what testing was done and how long it took, in order to improve any future test planning. This final document is used to indicate whether the software system under test is fit for purpose according to whether or not it has met acceptance criteria defined
-

by project stakeholders.

Relationship with other standards

Other standards that may be referred to when documenting according to IEEE 829 include:

- **IEEE 1008**, a standard for unit testing
- **IEEE 1012**, a standard for Software Verification and Validation
- **IEEE 1028**, a standard for software inspections
- **IEEE 1044**, a standard for the classification of software anomalies
- **IEEE 1044-1**, a guide to the classification of software anomalies
- **IEEE 830**, a guide for developing system requirements specifications
- **IEEE 730**, a standard for software quality assurance plans
- **IEEE 1061**, a standard for software quality metrics and methodology
- **IEEE 12207**, a standard for software life cycle processes and life cycle data
- **BS 7925-1**, a vocabulary of terms used in software testing
- **BS 7925-2**, a standard for software component testing

Use of IEEE 829

The standard forms part of the training syllabus of the ISEB Foundation and Practitioner Certificates in Software Testing promoted by the British Computer Society. ISTQB, following the formation of its own syllabus based on ISEB's and Germany's ASQF syllabi, also adopted IEEE 829 as the reference standard for software testing documentation.

Revisions

A revision to IEEE 829-1998, known as IEEE 829-2008 ^[1], was published on 18th July 2008 and when approved will supersede the 1998 version.

External links

- BS7925-2 ^[2], *Standard for Software Component Testing*
- [3] - IEEE Std 829-1998 (from IEEE)
- [4] - IEEE Std 829-2008 (from IEEE)
- [5] - IEEE Std 829-1998 (wilma.vub.ac.be)

References

- [1] <http://ieeexplore.ieee.org/Xplore/login.jsp?url=/ielD/4459216/4459217/04459218.pdf?arnumber=4459218>
- [2] <http://www.ruleworks.co.uk/testguide/BS7925-2.htm>
- [3] <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=741968&isnumber=16010>
- [4] <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4578383&isnumber=4578382>
- [5] http://wilma.vub.ac.be/~se1_0607/svn/bin/cgi/viewvc.cgi/documents/standards/IEEE/IEEE-STD-829-1998.pdf?revision=45

Independent software verification and validation

ISVV stands for **Independent Software Verification and Validation**. ISVV is targeted at safety-critical software systems and aims to increase the quality of software products, thereby reducing risks and costs through the operational life of the software. ISVV provides assurance that software performs to the specified level of confidence and within its designed parameters and defined requirements.

ISVV activities are performed by independent engineering teams, not involved in the software development process, to assess the processes and the resulting products. The ISVV team independency is performed at three different levels: financial, managerial and technical.

ISVV goes far beyond “traditional” verification and validation techniques, applied by development teams. While the latter aim to ensure that the software performs well against the nominal requirements, ISVV is focused on non-functional requirements such as robustness and reliability, and on conditions that can lead the software to fail. ISVV results and findings are fed back to the development teams for correction and improvement.

ISVV History

ISVV derives from the application of IV&V (Independent Verification and Validation) to the software. Early ISVV application (as known today) dates back to the early 1970s when the U.S. Army sponsored the first significant program related to IV&V for the Safeguard Anti-Ballistic Missile System.

By the end of the 1970s IV&V was rapidly becoming popular. The constant increase in complexity, size and importance of the software lead to an increasing demand on IV&V applied to software (ISVV).

Meanwhile IV&V (and ISVV for software systems) gets consolidated and is now widely used by organisations such as the DoD, FAA, NASA^[1] and ESA.^[2] IV&V is mentioned in [DO-178B], [ISO/IEC 12207] and formalised in [IEEE 1012].

Initially in 2004-2005, a European consortium led by the European Space Agency, and composed by DNV(N),^[3] Critical Software SA(P),^[4] Terma(DK)^[5] and CODA Scisys(UK)^[6] created the first version of a guide devoted to ISVV, called "ESA Guide for Independent Verification and Validation" with support from other organizations, eg SoftWcare SL (E) (^[7]), etc.

In 2008 the European Space Agency released a second version, being SoftWcare SL was the supporting editor having received inputs from many different European Space ISVV stakeholders. This guide covers the methodologies applicable to all the software engineering phases in what concerns ISVV.

ISVV Methodology

ISVV is usually composed by five principal phases, these phases can be executed sequentially or as results of a tailoring process.

ISVV Planning

- Planning of ISVV Activities
- System Criticality Analysis: Identification of Critical Components through a set of RAMS activities (Value for Money)
- Selection of the appropriate Methods and Tools

Requirements Verification

- Verification for: Completeness, Correctness, Testability

Design Verification

- Design adequacy and conformance to Software Requirements and Interfaces
-

- Internal and External Consistency
- Verification of Feasibility and Maintenance

Code Verification

- Verification for: Completeness, Correctness, Consistency
- Code Metrics Analysis
- Coding Standards Compliance Verification

Validation

- Identification of unstable components/functionalities
- Validation focused on Error-Handling: complementary (not concurrent!) validation regarding the one performed by the Development team (More for the Money, More for the Time)
- Compliance with Software and System Requirements
- Black box testing and White box testing techniques
- Experience based techniques

References

- [1] NASA IV&V Facility (<http://www.ivv.nasa.gov>)
- [2] ESA Web site (<http://www.esa.int>)
- [3] DNV Web site (<http://www.dnv.com>)
- [4] Critical Software SA Web site (<http://www.criticalsoftware.com>)
- [5] Terma Web site (<http://www.terma.com>)
- [6] Scisys Web site (<http://www.scisys.co.uk>)
- [7] SoftWcare SL Web site (<http://www.softwcare.com>)

Installation testing

Implementation testing installation testing is a kind of quality assurance work in the software industry that focuses on what customers will need to do to install and set up the new software successfully. The testing process may involve full, partial or upgrades install/uninstall processes.

This testing is typically done by the *software testing engineer* in conjunction with the configuration manager. Implementation testing is usually defined as testing which places a compiled version of code into the testing or pre-production environment, from which it may or may not progress into production. This generally takes place outside of the software development environment to limit code corruption from other future releases which may reside on the development network.

The simplest installation approach is to run an install program, sometimes called *package software*. This package software typically uses a setup program which acts as a multi-configuration wrapper and which may allow the software to be installed on a variety of machine and/or operating environments. Every possible configuration should receive an appropriate level of testing so that it can be released to customers with confidence.

In distributed systems, particularly where software is to be released into an already live target environment (such as an operational website) installation (or software deployment as it is sometimes called) can involve database schema changes as well as the installation of new software. Deployment plans in such circumstances may include back-out procedures whose use is intended to roll the target environment back if the deployment is unsuccessful. Ideally, the deployment plan itself should be tested in an environment that is a replica of the live environment. A factor that can increase the organizational requirements of such an exercise is the need to synchronize the data in the test deployment environment with that in the live environment with minimum disruption to live operation. This type of implementation may include testing of the processes which take place during the installation or upgrade of a multi-tier application. This type of testing is commonly compared to a dress rehearsal or may even be called a “dry

run”.

Integration testing

Integration testing (sometimes called Integration and Testing, abbreviated "I&T") is the phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before validation testing. Integration testing takes as its input modules that have been unit tested, groups them in larger aggregates, applies tests defined in an integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Purpose

The purpose of integration testing is to verify functional, performance, and reliability requirements placed on major design items. These "design items", i.e. assemblages (or groups of units), are exercised through their interfaces using Black box testing, success and error cases being simulated via appropriate parameter and data inputs. Simulated usage of shared data areas and inter-process communication is tested and individual subsystems are exercised through their input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing. The overall idea is a "building block" approach, in which verified assemblages are added to a verified base which is then used to support the integration testing of further assemblages.

Some different types of integration testing are big bang, top-down, and bottom-up.

Big Bang

In this approach, all or most of the developed modules are coupled together to form a complete software system or major part of the system and then used for integration testing. The Big Bang method is very effective for saving time in the integration testing process. However, if the test cases and their results are not recorded properly, the entire integration process will be more complicated and may prevent the testing team from achieving the goal of integration testing.

A type of Big Bang Integration testing is called **Usage Model testing**. Usage Model Testing can be used in both software and hardware integration testing. The basis behind this type of integration testing is to run user-like workloads in integrated user-like environments. In doing the testing in this manner, the environment is proofed, while the individual components are proofed indirectly through their use. Usage Model testing takes an optimistic approach to testing, because it expects to have few problems with the individual components. The strategy relies heavily on the component developers to do the isolated unit testing for their product. The goal of the strategy is to avoid redoing the testing done by the developers, and instead flesh out problems caused by the interaction of the components in the environment. For integration testing, Usage Model testing can be more efficient and provides better test coverage than traditional focused functional integration testing. To be more efficient and accurate, care must be used in defining the user-like workloads for creating realistic scenarios in exercising the environment. This gives that the integrated environment will work as expected for the target customers.

Top-down and Bottom-up

Bottom Up Testing is an approach to integrated testing where the lowest level components are tested first, then used to facilitate the testing of higher level components. The process is repeated until the component at the top of the hierarchy is tested.

All the bottom or low-level modules, procedures or functions are integrated and then tested. After the integration testing of lower level integrated modules, the next level of modules will be formed and can be used for integration testing. This approach is helpful only when all or most of the modules of the same development level are ready. This method also helps to determine the levels of software developed and makes it easier to report testing progress in the form of a percentage.

Top Down Testing is an approach to integrated testing where the top integrated modules are tested and the branch of the module is tested step by step until the end of the related module.

Sandwich Testing is an approach to combine top down testing with bottom up testing.

The main advantage of the Bottom-Up approach is that bugs are more easily found. With Top-Down, it is easier to find a missing branch link.

Limitations

Any conditions not stated in specified integration tests, outside of the confirmation of the execution of design items, will generally not be tested.

Integration Tree

An **Integration tree** (or **GUI tree**) is a graph that visualises all the GUI components of a software. Each node of the tree show the GUI components used in this software. The components on the leaves of the tree are modeless GUI elements, those on the nodes are modal GUI elements. GUIs are highly involved entities.^[1]

References

[1] The integration tree of *Microsoft WordPad* (http://www.cs.umd.edu/~atif/GUITARWeb/guitar_process_integration_tree.htm).

International Software Testing Qualifications Board

The **International Software Testing Qualifications Board** (ISTQB) is a software testing qualification certification organisation. Founded in Edinburgh in November 2002, ISTQB is a non-profit association legally registered in Belgium.

The ISTQB is responsible for the international qualification scheme called "ISTQB Certified Tester". The qualifications are based on a syllabus, and there is a hierarchy of qualifications and guidelines for accreditation and examination. The ISTQB is the world's leading issuer of Software Testing Qualifications having more 165000 certifications issued; the ISTQB consists of 47 member boards worldwide representing more than 60 countries (date: March 2011).

Currently ISTQB provides 3 levels of certification:

- Foundation Level
- Advanced Level
 - Test Manager
 - Test Analyst
 - Technical Test Analyst
- Expert Level
 - Improving the Test Process
 - Test Management
 - Test Automation (planned to be deployed in 2012)
 - Security Testing ((planned to be deployed in 2012)

The contents of each syllabus are taught as courses by training providers, which have been accredited by national or regional boards.. Each course is usually concluded by an examination covering the contents of the syllabus. After the examination, each successful participant receives the "ISTQB Certified Tester" certificate (or the local variant with the added "ISTQB compliant" logo). It is not compulsory to follow a course in order to attend the exam.

It is the ISTQB's goal to provide a single, universally accepted, international qualification scheme, aimed at software and system testing professionals, by providing the core syllabi and by setting guidelines for accreditation and examination, as well as auditing that the rules are observed by all the member boards. The ISTQB syllabi content has been used by thousands of organizations worldwide.

More detailed information can be found on the ISTQB Official Website ^[5].

Controversy

There is controversy surrounding the ISTQB's certification scheme. Many leaders in the software testing field do not recognise ISTQB certification as a worthwhile qualification^[1]

Despite some criticism, the ISTQB is today the most widespread and fast growing scheme for the certifications of competences on software testing; anybody interested in improving the ISTQB Syllabi and certification rules can apply to one of the National Boards and contribute to the several international ISTQB Working Groups that continuously enhance the ISTQB Body of knowledge, ensuring that all improvement opportunities are taken into account

External links

- [ISTQB Official Website](#) ^[5]

[1] <http://www.satisfice.com/blog/archives/36>

International Software Testing Qualifications Board Certified Tester

ISTQB Certified Tester is a standardized qualification for software testers. The certification is offered by the ISTQB (International Software Testing Qualifications Board), and is based on a multiple-choice exam covering the freely available syllabus. The education program is offered in over 47 countries (there are 47 Member Boards but some of them are regional, handling more than 1 country) (date: July 2010).

The ISTQB was officially founded as the International Software Testing Qualifications Board in Edinburgh in November 2002.

Certifications

- *Foundation Level (CTFL)*
- *Advanced Level - Test Manager*
- *Advanced Level - Test Analyst*
- *Advanced Level - Technical Test Analyst*
- *Advanced Level (CTAL) - Full Advanced Level (after passing the above exams of Advanced Level)*
- *Expert Level*

Content of the exams

The exam for the **Foundation Level** has a theoretical nature and requires knowledge of software development - especially in the field of software testing.

The different Advanced Level exams are more practical and require deeper knowledge in special areas. Test Manager deals with planning and control of the test process. Functional Tester concerns, amongst other things, reviews and Black box testing methods. Technical Tester includes component tests (also called *unit test*), requiring knowledge of White box testing and non-functional testing methods - this section also includes test tools. The Expert Level is still in preparation..

Pre-requisites

The Foundation Level exam has no pre-requisites. To take the Advanced Level exam, candidates need to pass the Foundation Level exam first and must prove professional experience (for example - the USA requirement is 60 months, India 24 months and Germany 18 months).

Exam

The exam consists of a multiple-choice test, which until 2005 about 80% of the candidates passed.

In USA exam fees for Foundation Level are \$250.00. For Advanced Level each component exam is USD 200.00 and there is a one-time qualification fee of USD 100.00 (date: September 2006).

Certification is valid for life (Foundation Level and Advanced Level) and there is no requirement for recertification.

Testing boards are responsible for the quality and the auditing of the examination. Worldwide there are testing boards in 47 countries (date: October 2010). In USA the corresponding organisation is the ASTQB (American Software Testing Qualifications Board).

Registering for Exams

Candidates can learn more about ISTQB certification and exams in their country by visiting the ISTQB website and the local boards' websites.

Figures about Certified Testers worldwide

As of June 2010 there were over 155,000 ISTQB Certified Testers, worldwide. One reason for this is that a number of organizations which offer other Software Testing certifications affiliate with ISTQB (e.g. ISEB (Information Systems Examination Board)).

Other Certifying Organisations

The QAI (Quality Assurance Institute) offers similar certification, in more than 40 countries.

Materials

Foundation Level

- Thomas Müller (chair), Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhäjärvi, Geoff Thompson and Erik van Veenendaal (2007) *Certified Tester - Foundation Level Syllabus - Version 2007*, International Software Testing Qualifications Board (ISTQB), Möhrendorf, Germany^[1]
- Andreas Spillner, Tilo Linz, Hans Schäfer (2006) *Software Testing Foundations - A Study Guide for the Certified Tester Exam - Foundation Level - ISTQB compliant*, 1st print. dpunkt.verlag GmbH, Heidelberg, Germany. ISBN 3-89864-363-8
- Andreas Spillner, Tilo Linz (2005) *Basiswissen Softwaretest - Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester: Foundation Level nach ISTQB-Standard*, 3rd extended and updated run, dpunkt.verlag GmbH, Heidelberg, Germany. ISBN 3-89864-358-1
- Brian Hambling (ed.) Peter Morgan, Geoff Thompson, Angelina Samaroo, Peter Williams (2006) *Software Testing: An ISEB Foundation*, British Computer Society, ISBN 978-1902505794
- Dorothy Graham, Erik van Veenendaal, Isabel Evans, and Rex Black. (2008). *Foundations of Software Testing*, Cengage, USA, ISBN 1844809897, ISBN 978-1844809899

Advanced Level

- German Testing Board e.V. (2003) *Software Testing - Advanced Level Syllabus - ISTQB-Certified-Tester, Advanced Level*, Version 1.2(E). European Organization for Quality - Software Group, Erlangen, Germany^[2]
- Erik van Veenendaal (ed. and author) (2005) *The Testing Practitioner*, 3rd run, UTN Publishers, CN Den Bosch, the Netherlands, ISBN 90-72194-65-9
- Andreas Spillner, Tilo Linz, Thomas Roßner, Mario Winter (2006) *Praxiswissen Softwaretest - Testmanagement: Aus- und Weiterbildung zum Certified Tester: Advanced Level nach ISTQB-Standard*, 1st run, dpunkt.verlag GmbH, Heidelberg, Germany, ISBN 3-89864-275-5
- Graham Bath / Judy McKay (2008) *The Software Test Engineer's Handbook - A Study Guide for the ISTQB Test Analyst and Technical Test Analyst Advanced Level Certificates*, Rocky Nook, USA; ISBN 978-1-933952-24-6

- Andreas Spillner / Tilo Linz / Thomas Rossner / Mario Winter (2007) *Software Testing Practice: Test Management*, A Study Guide for the Certified Tester Exam ISTQB Advanced Level, Rocky Nook, USA, ISBN: ISBN 978-1-933952-13-0

References

- [1] Certified Tester - Foundation Level Syllabus ([http://www.istqb.org/download/attachments/2326555/SyllabusFoundation\[1\].pdf](http://www.istqb.org/download/attachments/2326555/SyllabusFoundation[1].pdf)) International Software Testing Qualifications Board; PDF; 0.382 MB
- [2] Certified Tester - Advanced Level (<http://www.astqb.org/documents/ISTQBAdvancedLevelSyllabus.pdf>) International Software Testing Qualifications Board

External links

- ISTQB (<http://www.istqb.org>)
- American Software Testing Qualifications Board, Inc. (<http://www.astqb.org/>)
- Belgium and Dutch Testing Qualifications Board (<http://www.istqb.nl/>)
- iSQI (International Software Quality Institute) (<http://www.isqi.org/en/certification/certified-tester/>)

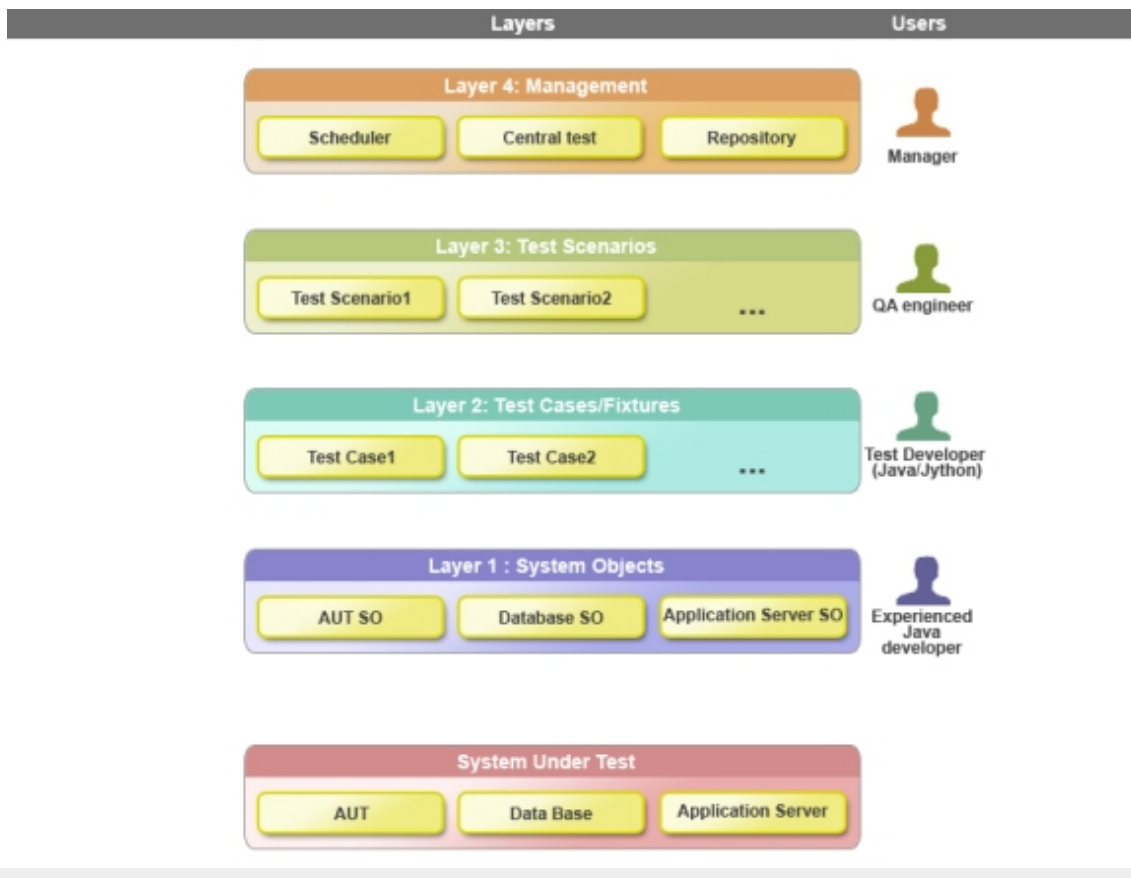
JSystem

JSystem is an open source framework for writing and running automated tests. The **JSystem Automation Framework** is written in Java and based on several open source java projects, using Eclipse as the development environment.

JSystem Automation Framework supports the full testing lifecycle, providing a solution for various types of users. JSystem enables the user to start with a small setup performing simple tests, and then enlarge the solution, providing a flexible solution for growing user needs.

The Four Layers of Automation

Automation projects written with JSystem are based on the **four layers of automation** architecture. JSystem divides the testing development process and architecture into four layers that simplifies and streamlines the development process; these layers are defined as follows:



SystemObject Layer

The first layer of JSystem is the SystemObject (also called Driver) layer. For every managed object in the system under test there is a SystemObject. The SystemObjects are written in Java and form the abstraction layer that controls the managed object.

SystemObject features and development guidelines

- **Hides Complexity** - This layer acts as a mediator that exposes the SUT to the Tests layer applying a simple user friendly interface that “speaks” in the language of the QA engineer. All the connectivity issues are hidden from the person who writes the tests.
- **Reporting** - Every SystemObject operation (method) leaves a remark, reporting its action. In some cases it collects information for analysis.
- **Error Handling** - If a step fails the test framework receives an exception directly circumventing the tests themselves.
- **Analyzes Results** - The SystemObject layer enables a “unite” mechanism to analyze results.

Tests Cases Layer

The Tests layer in JSystem is composed of tests and fixtures.

JSystem tests are based on <http://www.junit.org/tests>. Once the SystemObjects have been written, preparing the tests is a simple process, usually performed by a QA engineer with a programming orientation.

Fixtures are used to configure the SUT. Fixtures are Java classes that are responsible for bringing the system to the state required by the test. By using fixtures, the tests author can reuse configuration code and separate it from testing code.

Guidelines for writing tests

- **Keep it simple** - Do not complicate the test logic. Each test should be designed for a specific SUT. Do not design tests to run on a variety of SUT's.
- **Keep it short** - It is preferable to have a large number of shorter tests rather than a small number of longer, complicated tests.
- **Use fixtures** - Use fixtures to configure your SUT, and share the fixtures between tests.
- **Use Reports and Documentation** - If the tests are well documented, analyzing scenario execution results will be faster. It is crucial that you plan your project in such a way that it can scale to hundreds and thousands of tests.
- **Start with an Easy Test** - Start with simple and stable tests. Do not start to automate the most complex part of your system first.

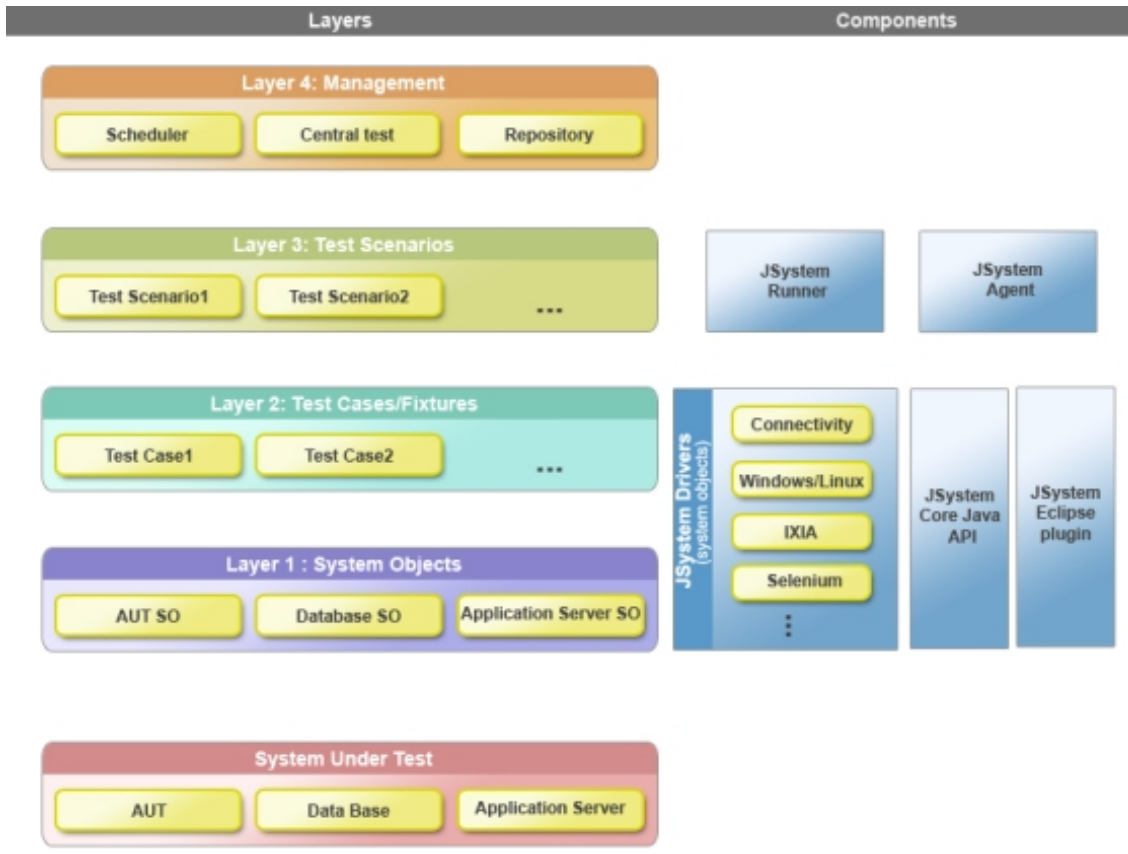
Test Scenarios Layer

Tests are grouped together in a hierarchical manner within a scenario. JSystem scenarios are written as Ant (<http://ant.apache.org/>) scripts. In addition to grouping the tests the user can parametrize a test, add flow control elements (if/else, for, switch/case), control the Java Virtual Machine (JVM) on which the test are activated, manage scenarios, execute scenarios, and analyze scenario execution results.

Management Layer

The Management layer comprises applications and services which purpose is to support enterprise development and execution of automation projects.

JSystem Components

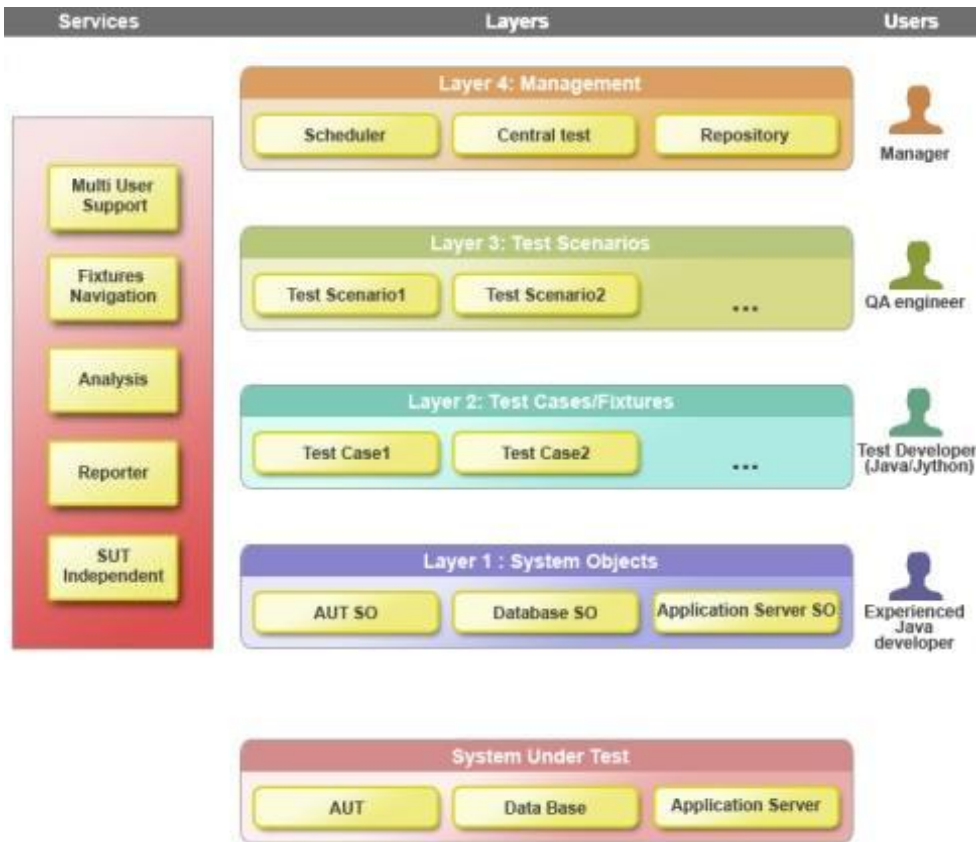


The JSystem framework is composed of the following components:

1. Services Java API - exposes JSystem services
2. JSystem Drivers (System Objects)- Java modules used to interfaces with devices and applications in the system under test.
3. JRunner - GUI application interface used for creating and running tests scenarios.
4. JSystem Agent - Execution engine used to run scenarios in a distributed setup.
5. JSystem Eclipse plug-in - accelerates the development environment setup and enforces JSystem conventions.

JSystem Framework Services

The JSystem Framework services refer to the set of API software applications provided by the JSystem Automation Framework that enable the user to develop and provide improved and streamlined project automation.



- **SUT Independence** - The SUT Independent Framework Service interacts directly with the system object. This service independence refers to the ability of the user to apply the same test to different DUT and SUT products and change parameters within the tests.
- **Reports**—The Reports Framework Service interacts directly with the system object layer and the test/fixtures layer, providing transparent information access to all other automation layers. When a scenario is run by JSystem the test case layer operates the system objects on the SUT or DUT the report framework service then extracts the results. These results are delivered to the reports framework service and are collated in the central management report mechanism called the JReporter application.
- **Analysis**—The Analysis Framework Service extracts statistics from the Report Framework Service connecting directly to results produced by the Test/Fixture layer. The Test/Fixture layer sends a request to the System Object layer for an analysis of a specific function; the results are then collated via the Report Framework Service and sent to the user.
- **Monitor**—The Monitor Framework Service runs a service that is performed in parallel to the test being performed. The Monitor Framework Services are written by the onsite Java programmer, according to the specification requirements of the product being tested.
- **Fixture**—The Fixture Framework Service connects directly to the Test/Fixture and the Scenario layers, similar the tests the fixtures are written by the on site Java programmer. The objective of the fixture is to bring the SUT to a state that enables the JSystem to perform tests. A fixture can be assigned to either a scenario or a Test/Fixture layer.
- **System Object Lifecycle**—The system object lifecycle service controls the initiation, system object state during execution and termination of the system object, and helps the user to implement a system object that works well

with the pause and graceful stop features de-allocating resources when the execution ends.

- **Multi User Support** - The Multi User Framework Service provides the test implementer a set of test variables that can be applied to a test or fixture via the scenario loaded the JSystem JRunner. This provides extended functionality and capabilities allowing the user to create and customize tests within the scenario by changing value options from within the “Test Info” tab. The user can construct variations to the test that sits inside the scenario. These selections are dynamic and intelligent, meaning that both the tasks and their content change dynamically according to the variable values chosen from the “Test Info” “Sub-Tab”.

JSystem Testing Approach

The JSystem approach to today’s testing requirements makes a clear distinction between the graphic user interface (GUI) and the business logic, testing the business logic as a separate entity from the user interface.

The JSystem solution is built on a committed API. By focusing on the business logic JSystem enables the user to take full advantage of the automated testing environment, providing increased testing stability and simplifying the testing project for the end user.

When dealing with an automation project JSystem assumes that the automation is a software project. The JSystem testing framework is augmented with a dynamic methodology and a robust architecture solution. The JSystem drivers enable the user to connect to multiple devices commonly found in a typical lab environment.

JSystem Key Focus

The central issue the JSystem Automation Framework solves is the maintenance aspect of the automation project providing the user tools to keep project maintenance to a minimum.

JSystem does this by focusing on four key aspects:

- **Maintainability**—JSystem enables the user to adjust the automation changes via a modulated system referred to as system objects (SystemObjects), These SystemObjects communicate directly to the business logic of the product. JSystem has the ability to connect directly to the application API enabling low maintenance migration.
- **Visibility**—JSystem provides a tool set that enables all user profiles the ability to easily interact with each other by clearly aligning the level of information they require. This stream lines the testing process by displaying the relevant information to each user profile.
- **Scalability**—The ability for a test project to grow in scale from ten’s of tests, to hundreds of tests to, thousands of tests. The JSystem application suite is built on an expandable code foundation that envisions project scaling and growth from the outset of the testing project.
- **Simplicity**—By clearly defining the user profiles level of understanding the system divides the test project into layers. These layers offer simplified environments for each user profile.

External Links

- Official website ^[1]

References

[1] <http://www.jssystemtest.org>

Keyword-driven testing

Keyword-driven testing, also known as **table-driven testing** or **action-word testing**, is a software testing methodology for automated testing that separates the test creation process into two distinct stages: a Planning Stage, and an Implementation Stage.

Overview

Although keyword testing can be used for manual testing, it is a technique particularly well suited to automated testing^[1]. The advantages for automated tests are the reusability and therefore ease of maintenance of tests that have been created at a high level of abstraction.

Methodology

The keyword-driven testing methodology divides test creation into two stages:-

- Planning Stage
- Implementation Stage

Planning Stage

Examples of keywords*

- A simple keyword (one action on one object), e.g. entering a username into a textfield.

Object	Action	Data
Textfield (username)	Enter text	<username>

- A more complex keyword (a combination of keywords into a meaningful unit), e.g. logging in.

Object	Action	Data
Textfield (domain)	Enter text	<domain>
Textfield (username)	Enter text	<username>
Textfield (password)	Enter text	<password>
Button (login)	Click	One left click

Implementation Stage

The implementation stage differs depending on the tool or framework. Often, automation engineers implement a framework that provides keywords like “check” and “enter”^[1]. Testers or test designers (who don’t have to know how to program) write test cases based on the keywords defined in the planning stage that have been implemented by the engineers. The test is executed using a driver that reads the keywords and executes the corresponding code.

Other methodologies use an all-in-one implementation stage. Instead of separating the tasks of test design and test engineering, the test design *is* the test automation. Keywords, such as “edit” or “check” are created using tools in which the necessary code has already been written. This removes the necessity for extra engineers in the test process, because the implementation for the keywords is already a part of the tool. Tools such as GUIDancer and QTP

Pros

1. Maintenance is low in a long run
 1. Test cases are concise
 2. Test Cases are readable for the stake holders
 3. Test Cases easy to modify
 4. New test cases can reuse existing keywords more easily
2. Keyword re-use across multiple test cases
3. Not dependent on Tool / Language
4. Division of Labor
 1. Test Case construction needs stronger domain expertise - lesser tool / programming skills
 2. Keyword implementation requires stronger tool/programming skill - with relatively lower domain skill
5. Abstraction of Layers.

Cons

1. Longer Time to Market (as compared to manual testing or record and replay technique)
2. Moderately high learning curve initially

References

- [1] (<http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=8186>), Danny R. Faught, Keyword-Driven Testing, Sticky Minds

External links

1. Hans Buwalda (http://www.logigear.com/newsletter/key_success_factors_for_keyword_driven_testing.asp), success factors for keyword driven testing.
2. SAFS (Software Automation Framework Support) (<http://safsdev.sourceforge.net>)
3. Test automation frameworks (<http://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.htm>)
4. Automation Framework - gFast: generic Framework for Automated Software Testing - QTP Framework (<http://www.slideshare.net/heydaysoft/g-fast-presentation/>)
5. Robot Framework Open Source Test Automation Framework (<http://robotframework.org>)

Learnability

Software testing

In software testing **learnability**, according to ISO/IEC 9126, is the capability of a software product to enable the user to learn how to use it. Learnability may be considered as an aspect of usability, and is of major concern in the design of complex software applications.

Learnability is defined in the *Standard glossary of terms used in software testing* published by the International Software Testing Qualifications Board.

Computational learning theory

In computational learning theory, **learnability** is the mathematical analysis of machine learning. It is also employed in language acquisition in arguments within linguistics.

Frameworks include:

- Language identification in the limit proposed in 1967 by E. Mark Gold.^[1] Subsequently known as Algorithmic learning theory.
- Probably approximately correct learning (PAC learning) proposed in 1984 by Leslie Valiant^[2]

References

[1] E.M. Gold. *Language identification in the limit* (<http://www.isrl.uiuc.edu/~amag/langev/paper/gold67limit.html>) Information and Control, 10(5), 1967.

[2] L. Valiant. *A theory of the learnable*. (<http://web.mit.edu/6.435/www/Valiant84.pdf>) Communications of the ACM, 27, 1984.

Lightweight software test automation

Lightweight software test automation is the process of creating and using relatively short and simple computer programs, called lightweight test harnesses, designed to test a software system. Lightweight test automation harnesses are not tied to a particular programming language but are most often implemented with the Java, Perl, Visual Basic .NET, and C# programming languages. Lightweight test automation harnesses are generally four pages of source code or less, and are generally written in four hours or less. Lightweight test automation is often associated with Agile software development methodology.

The three major alternatives to the use of lightweight software test automation are commercial test automation frameworks, Open Source test automation frameworks, and heavyweight test automation. The primary disadvantage of lightweight test automation is manageability. Because lightweight automation is relatively quick and easy to implement, a test effort can be overwhelmed with harness programs, test case data files, test result files, and so on. However, lightweight test automation has significant advantages. Compared with commercial frameworks, lightweight automation is less expensive in initial cost and is more flexible. Compared with Open Source frameworks, lightweight automation is more stable because there are fewer updates and external dependencies. Compared with heavyweight test automation, lightweight automation is quicker to implement and modify. Lightweight test automation is generally used to complement, not replace these alternative approaches.

Lightweight test automation is most useful for regression testing, where the intention is to verify that new source code added to the system under test has not created any new software failures. Lightweight test automation may be used for other areas of software testing such as performance testing, stress testing, load testing, security testing, code coverage analysis, mutation testing, and so on. The most widely published proponent of the use of lightweight software test automation is Dr. James D. McCaffrey.

References

- Definition and characteristics of lightweight software test automation in: McCaffrey, James D., ".NET Test Automation Recipes", Apress Publishing, 2006. ISBN: 1590596633.
 - Discussion of lightweight test automation versus manual testing in: Patton, Ron, "Software Testing, 2nd ed.", Sams Publishing, 2006. ISBN: 0672327988.
 - An example of lightweight software test automation for .NET applications: "Lightweight UI Test Automation with .NET", MSDN Magazine, January 2005 (Vol. 20, No. 1). See <http://msdn2.microsoft.com/en-us/magazine/cc163864.aspx>.
 - A demonstration of lightweight software test automation applied to stress testing: "Stress Testing", MSDN Magazine, May 2006 (Vol. 21, No. 6). See <http://msdn2.microsoft.com/en-us/magazine/cc163613.aspx>.
 - A discussion of lightweight software test automation for performance testing: "Web App Diagnostics: Lightweight Automated Performance Analysis", asp.netPRO Magazine, August 2005 (Vol. 4, No. 8).
 - An example of lightweight software test automation for Web applications: "Lightweight UI Test Automation for ASP.NET Web Applications", MSDN Magazine, April 2005 (Vol. 20, No. 4). See <http://msdn2.microsoft.com/en-us/magazine/cc163814.aspx>.
 - A technique for mutation testing using lightweight software test automation: "Mutant Power: Create a Simple Mutation Testing System with the .NET Framework", MSDN Magazine, April 2006 (Vol. 21, No. 5). See <http://msdn2.microsoft.com/en-us/magazine/cc163619.aspx>.
 - An investigation of lightweight software test automation in a scripting environment: "Lightweight Testing with Windows PowerShell", MSDN Magazine, May 2007 (Vol. 22, No. 5). See <http://msdn2.microsoft.com/en-us/magazine/cc163430.aspx>.
-

Load testing

Load testing is the process of putting demand on a system or device and measuring its response. Load testing is performed to determine a system's behavior under both normal and anticipated peak load conditions. It helps to identify the maximum operating capacity of an application as well as any bottlenecks and determine which element is causing degradation. When the load placed on the system is raised beyond normal usage patterns, in order to test the system's response at unusually high or peak loads, it is known as stress testing. The load is usually so great that error conditions are the expected result, although no clear boundary exists when an activity ceases to be a load test and becomes a stress test.

There is little agreement on what the specific goals of load testing are. The term is often used synonymously with software performance testing, reliability testing, and volume testing. *Load testing* is a type of non-functional testing.

Software load testing

The term *load testing* is used in different ways in the professional software testing community. *Load testing* generally refers to the practice of modeling the expected usage of a software program by simulating multiple users accessing the program concurrently. As such, this testing is most relevant for multi-user systems; often one built using a client/server model, such as web servers. However, other types of software systems can also be load tested. For example, a word processor or graphics editor can be forced to read an extremely large document; or a financial package can be forced to generate a report based on several years' worth of data. The most accurate load testing simulates actual use, as opposed to testing using theoretical or analytical modeling.

Load testing lets you measure your website's QOS performance based on actual customer behavior. Nearly all the load testing tools and frame-works follow the classical load testing paradigm, which is listed in Figure 1. When customers visit your web site, a script recorder records the communication and then creates related interaction scripts. A load generator tries to replay the recorded scripts, which could possibly be modified with different test parameters before replay. In the replay procedure, both the hardware and software statistics will be monitored and collected by the conductor, these statistics include the CPU, memory, disk IO of the physical servers and the response time, throughput of the System Under Test (short as SUT), etc. And at last, all these statistics will be analyzed and a load testing report will be generated.

Load and performance testing analyzes software intended for a multi-user audience by subjecting the software to different amounts of virtual and live users while monitoring performance measurements under these different loads. Load and performance testing is usually conducted in a test environment identical to the production environment before the software system is permitted to go live.

As an example, a web site with shopping cart capability is required to support 100 concurrent users broken out into following activities:

- 25 Virtual Users (VUsers) log in, browse through items and then log off
- 25 VUsers log in, add items to their shopping cart, check out and then log off
- 25 VUsers log in, return items previously purchased and then log off
- 25 VUsers just log in without any subsequent activity

A test analyst can use various load testing tools to create these VUsers and their activities. Once the test has started and reached a steady state, the application is being tested at the 100 VUser load as described above. The application's performance can then be monitored and captured.

The specifics of a load test plan or script will generally vary across organizations. For example, in the bulleted list above, the first item could represent 25 VUsers browsing unique items, random items, or a selected set of items depending upon the test plan or script developed. However, all load test plans attempt to simulate system performance across a range of anticipated peak workflows and volumes. The criteria for passing or failing a load test

(pass/fail criteria) are generally different across organizations as well. There are no standards specifying acceptable load testing performance metrics.

A common misconception is that load testing software provides record and playback capabilities like regression testing tools. Load testing tools analyze the entire OSI protocol stack whereas most regression testing tools focus on GUI performance. For example, a regression testing tool will record and playback a mouse click on a button on a web browser, but a load testing tool will send out hypertext the web browser sends after the user clicks the button. In a multiple-user environment, load testing tools can send out hypertext for multiple users with each user having a unique login ID, password, etc.

The popular load testing tools available also provide insight into the causes for slow performance. There are numerous possible causes for slow system performance, including, but not limited to, the following:

- Application server(s) or software
- Database server(s)
- Network – latency, congestion, etc.
- Client-side processing
- Load balancing between multiple servers

Load testing is especially important if the application, system or service will be subject to a service level agreement or SLA.

User Experience Under Load test

In the example above, while the device under test (DUT) is under production load - 100 VUsers, run the target application. The performance of the target application here would be the User Experience Under Load. It describe how fast or slow the DUT responds, and how satisfied or how the user actually perceives performance.

Many performance testers are running this test, but they call it different names. This name was selected by the Panelists and many Performance Testers in 2011 Online Performance Summit by STP ^[1].

There are already many tools and frameworks available to do the load testing from both commercial and open source.

Load testing tools

Tool Name	Company Name	Notes
AppLoader	NRG Global	Load and Performance testing Solution. Automates tests on the GUI level of the application. Can be used for unit, integration, and regression testing as well. Licensed.
blitz.io ^[2]	Mu Dynamics	Blitz enables self-service load and performance testing for cloud and mobile applications. Solution is focused on continuous testing for DevOps that commonly make multiple changes every day.
IBM Rational Performance Tester	IBM	Eclipse based large scale performance testing tool primarily used for executing large volume performance tests to measure system response time for server based applications. Licensed.
IXIA IxLoad	Ixia (company)	Chassis based Load and Performance Testing System with High Performance (10G/40G/100G-Multiport Cards). Licensed.
JMeter	An Apache Jakarta open source project	Java desktop application for load testing and performance measurement.
Load Test (included with Soatest)	Parasoft	Performance testing tool that verifies functionality and performance under load. Supports SOAtest tests, JUnits, lightweight socket-based components. Detects concurrency issues.
LoadRunner	HP	Performance testing tool primarily used for executing large numbers of tests (or a large number of virtual users) concurrently. Can be used for unit and integration testing as well. Licensed.

OpenSTA	Open System Testing Architecture	Open source web load/stress testing application, licensed under the Gnu GPL. Utilizes a distributed software architecture based on CORBA. OpenSTA binaries available for Windows.
SilkPerformer	Micro Focus	Performance testing in an open and sharable model which allows realistic load tests for thousands of users running business scenarios across a broad range of enterprise application environments.
SLAMD		Open source, 100% Java web application, scriptable, distributed with Tomcat.
Visual Studio Load Test	Microsoft	Visual Studio includes a load test tool which enables a developer to execute a variety of tests (web, unit etc...) with a combination of configurations to simulate real user load. ^[3]

Mechanical load testing

The purpose of a mechanical load test is to verify that all the component parts of a structure including materials, base-fixings are fit for task and loading it is designed for.

The *Supply of Machinery (Safety) Regulation 1992 UK* state that load testing is undertaken before the equipment is put into service for the first time.

Load testing can be either **Performance,Static** or **Dynamic**.

Performance testing is when the stated safe working load (SWL) for a configuration is used to determine that the item performs to the manufactures specification. If an item fails this test then any further tests are pointless.

Static testing is when a load at a factor above the SWL is applied. The item is not operated through all configurations as it is not a requirement of this test.

Dynamic testing is when a load at a factor above the SWL is applied. The item is then operated fully through all configurations and motions. Care must be taken during this test as there is a great risk of catastrophic failure if incorrectly carried out.

The design criteria, relevant legislation or the *Competent Person* will dictate what test is required.

Under the *Lifting Operations and Lifting Equipment Regulations 1998 UK* load testing after the initial test is required if a major component is replaced, if the item is moved from one location to another or as dictated by the *Competent Person*

The loads required for a test are stipulated by the item under test, but here are a few to be aware off. Powered lifting equipment **Static** test to 1.25 SWL and **dynamic** test to 1.1 SWL. Manual lifting equipment **Static** test to 1.5 SWL

For lifting accessories. 2 SWL for items up to 30 tonne capacity. 1.5 SWL for items above 30 tonne capacity. 1 SWL for items above 100 tonnes.

Car charging system

A load test can be used to evaluate the health of a car's battery. The tester consists of a large resistor that has a resistance similar to a car's starter motor and a meter to read the battery's output voltage both in the unloaded and loaded state. When the tester is used, the battery's open circuit voltage is checked first. If the open circuit voltage is below spec (12.6 volts for a fully charged battery), the battery is charged first. After reading the battery's open circuit voltage, the load is applied. When applied, it draws approximately the same current the car's starter motor would draw during cranking. Based on the specified cold cranking amperes of the battery, if the voltage under load falls below a certain point, the battery is bad. Load tests are also used on running cars to check the output of the car's alternator.

References

[1] <http://www.softwaretestpro.com/>

[2] <http://blitz.io/>

[3] <http://www.eggheadcafe.com/tutorials/aspnet/13e16f83-4cf2-4c9d-b75b-aa67fc309108/load-testing-aspnet-appl.aspx>

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5634372&tag=1 **Supply of Machinery (Safety) Regulations 1992 UK. Lifting Operations and Lifting Equipment Regulations 1998 UK.**

External links

- Modeling the Real World for Load Testing Web Sites (<http://www.methodsandtools.com/archive/archive.php?id=38>) by Steven Splaine

Localization testing

Localization testing is a part of software testing process focused on internationalization and localization aspects of software. Localization is the process of adapting a globalized application to a particular culture/locale. Localizing an application requires a basic understanding of the character sets typically used in modern software development and an understanding of the issues associated with them. Localization includes the translation of the application user interface and adapting graphics for a specific culture/locale. The localization process can also include translating any help content associated with the application.

Localization of business solutions requires that you implement the correct business processes and practices for a culture/locale. Differences in how cultures/locales conduct business are heavily shaped by governmental and regulatory requirements. Therefore, localization of business logic can be a massive task.

Localization testing checks how well the build has been Localized into a particular target language. This test is based on the results of globalized testing where the functional support for that particular locale has already been verified. If the product is not globalized enough to support a given language, you probably will not try to localize it into that language in the first place!

You still have to check that the application you're shipping to a particular market really works and the following section shows you some of the general areas on which to focus when performing a localization test.

The following needs to be considered in localization testing:

- Things that are often altered during localization, such as the UserInterface and content files.
 - Operating System
 - Keyboards
 - Text Filters
 - Hot keys
 - Spelling Rules
 - Sorting Rules
 - Upper and Lower case conversions
 - Printers
 - Size of Papers
 - Mouse
 - Date formats
 - Rulers and Measurements
 - Memory Availability
 - Voice User Interface language/accent
 - Video Content
-

It's also a good idea to check that everything you are going to distribute in a local market complies with the local laws and regulations.

Manual testing

Compare with Test automation.

Manual testing is the process of manually testing software for defects. It requires a tester to play the role of an end user, and use most of all features of the application to ensure correct behavior. To ensure completeness of testing, the tester often follows a written test plan that leads them through a set of important test cases.

Overview

A key step in the process of software engineering is testing the software for correct behavior prior to release to end users.

For small scale engineering efforts (including prototypes), exploratory testing may be sufficient. With this informal approach, the tester does not follow any rigorous testing procedure, but rather explores the user interface of the application using as many of its features as possible, using information gained in prior tests to intuitively derive additional tests. The success of exploratory manual testing relies heavily on the domain expertise of the tester, because a lack of knowledge will lead to incompleteness in testing. One of the key advantages of an informal approach is to gain an intuitive insight to how it feels to use the application.

Large scale engineering projects that rely on manual software testing follow a more rigorous methodology in order to maximize the number of defects that can be found. A systematic approach focuses on predetermined test cases and generally involves the following steps.^[1]

1. Choose a high level test plan where a general methodology is chosen, and resources such as people, computers, and software licenses are identified and acquired.
2. Write detailed test cases, identifying clear and concise steps to be taken by the tester, with expected outcomes.
3. Assign the test cases to testers, who manually follow the steps and record the results.
4. Author a test report, detailing the findings of the testers. The report is used by managers to determine whether the software can be released, and if not, it is used by engineers to identify and correct the problems.

A rigorous test case based approach is often traditional for large software engineering projects that follow a Waterfall model.^[2] However, at least one recent study did not show a dramatic difference in defect detection efficiency between exploratory testing and test case based testing.^[3]

Stages

There are several stages. They are

Unit Testing This initial stage in testing normally carried out by the developer who wrote the code and sometimes by a peer using the white box testing technique.

Integration Testing This stage is carried out in two modes, as a complete package or as an increment to the earlier package. Most of the time black box testing technique is used. However, sometimes a combination of Black and White box testing is also used in this stage.

System Testing In this stage the software is tested from all possible dimensions for all intended purposes and platforms. In this stage Black box testing technique is normally used.

User Acceptance Testing This testing stage carried out in order to get customer sign-off of finished product. A 'pass' in this stage also ensures that the customer has accepted the software and is ready for their use.

[4]

Comparison to Automated Testing

Test automation may be able to reduce or eliminate the cost of actual testing. A computer can follow a rote sequence of steps more quickly than a person, and it can run the tests overnight to present the results in the morning. However, the labor that is saved in actual testing must be spent instead authoring the test program. Depending on the type of application to be tested, and the automation tools that are chosen, this may require more labor than a manual approach. In addition, some testing tools present a very large amount of data, potentially creating a time consuming task of interpreting the results. From a cost-benefit perspective, test automation becomes more cost effective when the same tests can be reused many times over, such as for regression testing and test-driven development, and when the results can be interpreted quickly. If future reuse of the test software is unlikely, then a manual approach is preferred.^[5]

Things such as device drivers and software libraries must be tested using test programs. In addition, testing of large numbers of users (performance testing and load testing) is typically simulated in software rather than performed in practice.

Conversely, graphical user interfaces whose layout changes frequently are very difficult to test automatically. There are test frameworks that can be used for regression testing of user interfaces. They rely on recording of sequences of keystrokes and mouse gestures, then playing them back and observing that the user interface responds in the same way every time. Unfortunately, these recordings may not work properly when a button is moved or relabeled in a subsequent release. An automatic regression test may also be fooled if the program output varies significantly (e.g. the display includes the current system time). In cases such as these, manual testing may be more effective.^[6]

References

- [1] ANSI/IEEE 829-1983 IEEE Standard for Software Test Documentation
- [2] Craig, Rick David; Stefan P. Jaskiel (2002). *Systematic Software Testing*. Artech House. p. 7. ISBN 1580535089.
- [3] Itkonen, Juha; Mika V. Mäntylä and Casper Lassenius (2007). "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing" (http://www.soberit.hut.fi/jitkonen/Publications/Itkonen_Mäntylä_Lassenius_2007_ESEM.pdf). *First International Symposium on Empirical Software Engineering and Measurement*. . Retrieved 2009-01-17.
- [4] <http://softwaretestinginterviewfaqs.wordpress.com/category/testing-in-stages/>
- [5] Mosley, Daniel (2002). *Just Enough Software Test Automation*. Prentice Hall. p. 27. ISBN 0130084689.
- [6] Bach, James (1996). "Test Automation Snake Oil" (http://www.satisfice.com/articles/test_automation_snake_oil.pdf). *Windows Technical Journal* **10/96**: 40–44. . Retrieved 2009-01-17.

Matrix of Pain

In software development, the **Matrix of Pain** refers to the array of potential customer configurations for which developers must test their software. A significant amount of resources can be spent ensuring that software will operate on a seemingly endless array of potential end-user environments which include various hardware configurations, operating system types, operating system versions, as well as other forms of system software. A tradeoff involved with this process is that resources spent trying to anticipate and account for all of these combinations are at the expense of new feature development and/or refinement of existing feature functionality.

References

- Beyond Software Architecture: Creating and Sustaining Winning Solutions ^[1], Jan 30, 2003, ISBN 0-201-77594-8

References

[1] <http://books.google.com/books?vid=ISBN0201775948>

Mauve (test suite)

Mauve

Operating system	Java virtual machine
Type	Test Suite
License	GNU General Public License
Website	[1]

Mauve is a project to provide a free software test suite for the Java class libraries. Mauve is developed by the members of Kaffe, GNU Classpath, GCJ, and other projects. Unlike a similar project, JUnit, Mauve is designed to run on various experimental Java virtual machines, where some features may be still missing. Because of this, Mauve does not discover the testing method by name, as JUnit does. Mauve can also be used to test the user java application, not just the core class library. Mauve is released under GNU General Public License.

Example

The "Hello world" example in Mauve:

```
//Tags: JDK1.4
public class HelloWorld implements Testlet {
    // Test if 3*2=6
    public void test(TestHarness harness) {
        harness.check(3*2, 6, "Multiplication failed.");
    }
}
```

External links

- [Mauve homepage](#) ^[1]

References

- [1] <http://www.sourceforge.org/mauve/>

Metasploit Project

Metasploit Framework

<pre>msf exploit(windows/dcerp [*] Started reverse handl [*] Trying target Windows [*] Binding to 4d9f4ab8-7 [*] Bound to 4d9f4ab8-7d1 [*] sending exploit ... [*] Sending stage (2834 b [*] Sleeping before handl [*] Uploading DLL (73739 [*] Upload completed. [*] Meterpreter session 1 Loading extension stdapi. meterpreter > use priv Loading extension priv... meterpreter > hashdump Administrator:5000: "Point. Click. Root."</pre>	
Developer(s)	Rapid7 LLC
Stable release	4.0 / August 1, 2011
Development status	Active
Operating system	Cross-platform
Type	Security
License	BSD
Website	http://www.metasploit.com/

The **Metasploit Project** is an open-source computer security project which provides information about security vulnerabilities and aids in penetration testing and IDS signature development. Its most well-known sub-project is the **Metasploit Framework**, a tool for developing and executing exploit code against a remote target machine. Other important sub-projects include the Opcode Database, shellcode archive, and security research.

The Metasploit Project is also well-known for anti-forensic and evasion tools, some of which are built into the Metasploit Framework.

Metasploit was created by HD Moore in 2003 as a portable network tool using the Perl scripting language. Later, the Metasploit Framework was then completely rewritten in the Ruby programming language.^[1] In addition, it is a tool for third-party security researchers to investigate potential vulnerabilities. On October 21, 2009 the Metasploit Project announced^[2] that it had been acquired by Rapid7, a security company that provides unified vulnerability management solutions.

Like comparable commercial products such as Immunity's Canvas or Core Security Technologies' Core Impact, Metasploit can be used to test the vulnerability of computer systems to protect them, and it can be used to break into remote systems. Like many information security tools, Metasploit can be used for both legitimate and unauthorized activities. Since the acquisition of the Metasploit Framework, Rapid7 has added two open core proprietary editions called Metasploit Express and Metasploit Pro.

Metasploit's emerging position as the de facto exploit development framework^[3] has led in recent times to the release of software vulnerability advisories often accompanied by a third party Metasploit exploit module that highlights the exploitability, risk, and remediation of that particular bug.^[4] ^[5] Metasploit 3.0 (Ruby language) is also beginning to include fuzzing tools, to discover software vulnerabilities, rather than merely writing exploits for currently public bugs. This new avenue has been seen with the integration of the lorcon wireless (802.11) toolset into Metasploit 3.0 in November 2006. Metasploit 4.0 was released in August 2011.

Metasploit Framework

The basic steps for exploiting a system using the Framework include -

1. Choosing and configuring an *exploit* (code that enters a target system by taking advantage of one of its bugs; about 300 different exploits for Windows, Unix/Linux and Mac OS X systems are included);
2. Checking whether the intended target system is susceptible to the chosen exploit (optional);
3. Choosing and configuring a *payload* (code that will be executed on the target system upon successful entry, for instance a remote shell or a VNC server);
4. Choosing the encoding technique to encode the payload so that the intrusion-prevention system (IPS) will not catch the encoded payload;
5. Executing the exploit.

This modularity of allowing to combine any exploit with any payload is the major advantage of the Framework: it facilitates the tasks of attackers, exploit writers, and payload writers.

Versions of the Metasploit Framework since v3.0 are written in the Ruby programming language. The previous version 2.7, was implemented in Perl. It runs on all versions of Unix (including Linux and Mac OS X), and also on Windows. It includes two command-line interfaces, a web-based interface and a native GUI. The web interface is intended to be run from the attacker's computer. The Metasploit Framework can be extended to use external add-ons in multiple languages.

To choose an exploit and payload, some information about the target system is needed such as operating system version and installed network services. This information can be gleaned with port scanning and OS fingerprinting tools such as nmap. Vulnerability scanners such as NeXpose or Nessus can detect the target system vulnerabilities. Metasploit can import vulnerability scan data and compare the identified vulnerabilities to existing exploit modules for accurate exploitation.

Metasploit Express

In April 2010, Rapid7 released Metasploit Express, an open-core commercial edition for security teams who need to verify vulnerabilities.^[6] Built on the Metasploit Framework, it offers a graphical user interface, integrates nmap for discovery, and adds smart bruteforcing as well as automated evidence collection.^[7] Rapid7 offers a 7-day trial for Metasploit Express.^[8]

Metasploit Pro

In October 2010, Rapid7 added Metasploit Pro, an open-core commercial Metasploit edition for penetration testers.^[9] Metasploit Pro includes all features of Metasploit Express and adds web application scanning and exploitation, social engineering campaigns, and VPN pivoting.^[10] Metasploit Pro is available as a 7-day trial.^[11]

Payloads

Metasploit offers many types of payloads, including:

- **Command shell** enables users to run collection scripts or run arbitrary commands against the host.
- **Meterpreter** enables users to control the screen of a device using VNC and to browse, upload and download files.

Opcode Database

The Opcode Database is an important resource for writers of new exploits. Buffer overflow exploits on Windows often require precise knowledge of the position of certain machine language opcodes in the attacked program or included DLLs. These positions differ in the various versions and patch-levels of a given operating system, and they are all documented and conveniently searchable in the Opcode Database. This allows one to write buffer overflow exploits which work across different versions of the target operating system.

Shellcode Database

The Shellcode database contains the payloads (also known as shellcode) used by the Metasploit Framework. These are written in assembly language and full source code is available.

References

- [1] Metasploit History (<http://www.metasploit.com/learn-more/history/>)
- [2] Rapid7 Acquires Metasploit (<http://www.rapid7.com/metasploit-announcement.jsp>)
- [3] Sctools.org survey showing Metasploit as #1 exploit tool (<http://sectools.org/splouts.html>)
- [4] "ACSSEC-2005-11-25-0x1 VMWare Workstation 5.5.0 <= build-18007 GSX Server Variants And Others" (<http://archives.neohapsis.com/archives/vulnwatch/2005-q4/0074.html>). December 20, 2005. .
- [5] "Month of Kernel Bugs - Broadcom Wireless Driver Probe Response SSID Overflow" (<http://projects.info-pull.com/mokb/MOKB-11-11-2006.html>). November 11, 2006. .
- [6] Press Release: Rapid7 Takes Penetration Testing Mainstream With Metasploit Express (<http://www.rapid7.com/news-events/press-releases/2010/2010-metasploit-express.jsp>)
- [7] Metasploit Express product page (<http://www.rapid7.com/products/metasploit-express/>)
- [8] Metasploit Express trial registration (<http://www.rapid7.com/downloads/metasploit-express.jsp>)
- [9] Rapid7 Introduces Metasploit Pro - The World's First Penetration Testing Solution that Achieves Unrestricted Remote Network Access Through Firewalls (<http://www.rapid7.com/news-events/press-releases/2010/2010-introduces-metasploit-pro.jsp>)
- [10] Metasploit Pro product page (<http://www.rapid7.com/products/metasploit-pro.jsp>)
- [11] Metasploit Pro trial registration (<http://www.rapid7.com/downloads/metasploit-pro.jsp>)

Further reading

- *Powerful payloads: The evolution of exploit frameworks* (http://searchsecurity.techtarget.com/originalContent/0,289142,sid14_gci1135581,00.html), searchsecurity.com, 2005-10-20
- Chapter 12: Writing Exploits III from *Sockets, Shellcode, Porting & Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals* by James C. Foster (ISBN 1-59749-005-9). Written by Vincent Liu, chapter 12 explains how to use Metasploit to develop a buffer overflow exploit from scratch.
- *HackMiami Pwn-Off Hack-A-Thon review of Metasploit Express* (<http://www.n00bz.net/metasploit-express/>)

External links

- The Metasploit Project (<http://www.metasploit.com/>) official website
- Metasploit Community (<https://community.rapid7.com/community/metasploit>) – The Official Metasploit online community
- Metasploit Express (<http://www.rapid7.com/products/metasploit-express.jsp>) commercial open-core Metasploit edition for security teams
- Metasploit Pro (<http://www.rapid7.com/products/metasploit-pro.jsp>) commercial open-core Metasploit edition for penetration testers
- Rapid7 LLC (<http://www.rapid7.com/>) owner of the Metasploit Project
- Metasploit Framework (<http://freshmeat.net/projects/msf/>) at Freshmeat
- Metasploit Framework (<http://www.ohloh.net/p/metasploit/>) at Ohloh
- Metasploit Unleashed - Mastering The Framework (<http://www.offensive-security.com/metasploit-unleashed/>)

Microsoft Reaction Card Method (Desirability Testing)

Developed by Microsoft in 2002 by Joey Benedek and Trish Miner, the **Microsoft Reaction Card** method is used to check the emotional response and desirability of a design or product. This method is commonly used in the field of software design.

The participant is asked to describe a design / product using any number of the following 118 words:

1. Accessible
 2. Advanced
 3. Annoying
 4. Appealing
 5. Approachable
 6. Attractive
 7. Boring
 8. Business-like
 9. Busy
 10. Calm
 11. Clean
 12. Clear
 13. Collaborative
 14. Comfortable
 15. Compatible
 16. Compelling
 17. Complex
 18. Comprehensive
 19. Confident
 20. Confusing
 21. Connected
 22. Consistent
 23. Controllable
 24. Convenient
 25. Creative
-

26. Customizable
 27. Cutting edge
 28. Dated
 29. Desirable
 30. Difficult
 31. Disconnected
 32. Disruptive
 33. Distracting
 34. Dull
 35. Easy to use
 36. Effective
 37. Efficient
 38. Effortless
 39. Empowering
 40. Energetic
 41. Engaging
 42. Entertaining
 43. Enthusiastic
 44. Essential
 45. Exceptional
 46. Exciting
 47. Expected
 48. Familiar
 49. Fast
 50. Flexible
 51. Fragile
 52. Fresh
 53. Friendly
 54. Frustrating
 55. Fun
 56. Gets in the way
 57. Hard to Use
 58. Helpful
 59. High quality
 60. Impersonal
 61. Impressive
 62. Incomprehensible
 63. Inconsistent
 64. Ineffective
 65. Innovative
 66. Inspiring
 67. Integrated
 68. Intimidating
 69. Intuitive
 70. Inviting
 71. Irrelevant
 72. Low Maintenance
-

73. Meaningful
 74. Motivating
 75. Not Secure
 76. Not Valuable
 77. Novel
 78. Old
 79. Optimistic
 80. Ordinary
 81. Organized
 82. Overbearing
 83. Overwhelming
 84. Patronizing
 85. Personal
 86. Poor quality
 87. Powerful
 88. Predictable
 89. Professional
 90. Relevant
 91. Reliable
 92. Responsive
 93. Rigid
 94. Satisfying
 95. Secure
 96. Simplistic
 97. Slow
 98. Sophisticated
 99. Stable
 100. Sterile
 101. Stimulating
 102. Straight Forward
 103. Stressful
 104. Time-consuming
 105. Time-Saving
 106. Too Technical
 107. Trustworthy
 108. Unapproachable
 109. Unattractive
 110. Uncontrollable
 111. Unconventional
 112. Understandable
 113. Undesirable
 114. Unpredictable
 115. Unrefined
 116. Usable
 117. Useful
 118. Valuable
-

Each word is placed on a separate card. After viewing a design or product the participant is asked to pick out the words they feel are relevant. The moderator would then ask the participant to describe their rationale for their selection.

References

- <http://uxmatters.com/mt/archives/2010/02/rapid-desirability-testing-a-case-study.php> - online source of information
- <http://www.mojoleaf.com> - remote testing incorporating the Microsoft Reaction Card Method

Mobile Device Testing

Mobile Device Testing is the process to assure the quality of mobile devices, like mobile phones, PDAs, etc. The testing will be conducted on both hardware and software. And from the view of different procedures, the testing comprises R&D Testing, Factory Testing and Certificate Testing.

Mobile device testing involves a set of activities from monitoring and trouble shooting mobile application, content and services on real handsets. Testing includes verification and validation of hardware devices and software applications.

Listed companies like Keynote systems provide mobile testing helping developers and mobile device manufacturers in testing and monitoring of mobile content, applications and services.^[1]

Static Code Analysis

Static code analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis)^[2] Static analysis rules are available for code written to target various mobile development platforms.

Unit Testing

Unit testing is a test phase when portions of mobile device development are testing typically by the developer. It may contain **hardware testing**, **software testing**, and **mechanical testing**.

Factory Testing

Factory Testing is a kind of sanity check on mobile devices. It's conducted automatically to verify that there are no defects brought by the manufacturing or assembling.

Mobile Testing contains

Mobile application testing Hardware testing

Battery (Charging) Testing
Signal receiving
Network Testing

Protocol testing mobile games Testing Mobile software compatibility Testing.

Certification Testing

Certification Testing is the check before a mobile device goes to market. Many institutes or governments require mobile devices to conform with their stated specifications and protocols to make sure the mobile device will not harm users' health and are compatible with devices from other manufacturers. Once the mobile device passes all checks, a certification will be issued for it.

Certification Forums

PTCRB, GCF

References

- [1] Mobile Monitoring and Testing (http://www.keynote.com/products/mobile_quality/index.html)
- [2] Industrial Perspective on Static Analysis. Software Engineering Journal Mar. 1995: 69-75Wichmann, B. A., A. A. Canning, D. L. Clutterbuck, L. A. Winsbarrow, N. J. Ward, and D. W. R. Marsh. <http://www.ida.liu.se/~TDDC90/papers/industrial95.pdf>

Mockito

Mockito is an open source testing framework for Java released under the MIT License. The framework allows the creation of Test Double objects called, "Mock Objects" in automated unit tests for the purpose of Test-driven Development (TDD) or Behavior Driven Development (BDD).

Distinguishing Features

Mockito distinguishes itself from other mocking frameworks by allowing developers to verify the behavior of the system under test (SUT) without establishing expectations beforehand^[1]. One of the criticisms of Mock objects is that there is a tighter coupling of the test code to the SUT object code^[2]. Since **Mockito** attempts to eliminate the expect-run-verify pattern^[3] by removing the specification of expectations, the coupling is reduced or minimized. The result of this distinguishing feature is simpler test code that should be easier to read and modify.

Origins

Szczepan Faber started the **Mockito** project after finding existing mocking frameworks too complex and difficult to work with. Szczepan began by expanding on the syntax and functionality of Easy Mock, but eventually rewriting most of **Mockito**^[4]. Szczepan's goal was to create a new framework that was easier to work with and provided better results. Early versions of **Mockito** project found use by the Guardian project in London in early 2008^[5].

Usage

Mockito has a growing user-base^[6] as well as finding use in other open source projects^[7]. In the Stack Overflow discussion on *What's the best mock framework for Java?*, **Mockito** is the highest recommended answer^[8].

See Also

- List of mock object frameworks
- Behavior driven development
- Mock object
- List of unit testing frameworks
- Software testing
- Unit testing

References

- [1] "*Features and Motivations*" (<http://code.google.com/p/mockito/wiki/FeaturesAndMotivations>). . Retrieved 2010-12-29.
- [2] Fowler, Martin (2007). "*Mocks Aren't Stubs*" (<http://martinfowler.com/articles/mocksArentStubs.html#CouplingTestsToImplementations>). . Retrieved 2010-12-29.
- [3] Faber, Szczepan. "*Death Wish*" (<http://monkeyisland.pl/2008/02/01/deathwish/>). . Retrieved 2010-12-29.
- [4] Faber, Szczepan. "*Mockito*" (<http://monkeyisland.pl/2008/01/14/mockito/>). . Retrieved 2010-12-29.
- [5] "Mockito Home Page" (<http://code.google.com/p/mockito/>). . Retrieved 2010-12-29.
- [6] "*Mockito User Base*" (<http://code.google.com/p/mockito/wiki/UserBase>). . Retrieved 2010-12-29.
- [7] "*Mockito in Use*" (<http://code.google.com/p/mockito/wiki/MockitoInUse>). . Retrieved 2010-12-29.
- [8] "*What's the best mock framework for Java?*" (<http://stackoverflow.com/questions/22697/whats-the-best-mock-framework-for-java>). . Retrieved 2010-12-29.

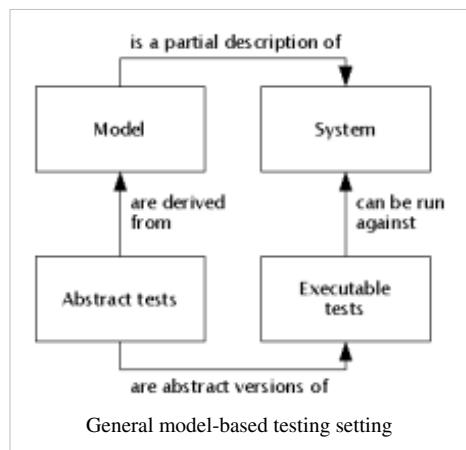
External links

- <http://mockito.org/> (<http://mockito.org/>)
- <http://easymock.org/> (<http://easymock.org/>)

Model-based testing

Model-based testing is the application of Model based design for designing and executing the necessary artifacts to perform software testing. This is achieved by having a model that describes all aspects of the testing data, mainly the test cases and the test execution environment. Usually, the testing model is derived in whole or in part from a model that describes some (usually functional) aspects of the system under development (SUD).

The model describing the SUD is usually an abstract, partial presentation of the system under test's desired behavior. The test cases derived from this model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. The abstract test suite cannot be directly executed against the system under test because it is on the wrong level of abstraction. Therefore an executable test suite must be derived from the abstract test suite that can communicate with the system under test. This is done by mapping the abstract test cases to concrete test cases suitable for execution. In some model-based testing tools, the model contains enough information to generate an executable test suite from it. In the case of online testing (see below), the abstract test suite exists only as a concept but not as an explicit artifact.



There are many different ways to "derive" tests from a model. Because testing is usually experimental and based on heuristics, there is no one best way to do this. It is common to consolidate all test derivation related design decisions into a package that is often known as "test requirements", "test purpose" or even "use case". This package can

contain e.g. information about the part of the model that should be the focus for testing, or about the conditions where it is correct to stop testing (test stopping criteria).

Because test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing. In some aspects, this is not completely accurate. Model-based testing can be combined with source-code level test coverage measurement, and functional models can be based on existing source code in the first place.

Model-based testing for complex software systems is still an evolving field.

Models

Especially in Model Driven Engineering or in OMG's model-driven architecture the model is built before or parallel to the development process of the system under test. The model can also be constructed from the completed system. Recently the model is created mostly manually, but there are also attempts to create the model automatically, for instance out of the source code. One important way to create new models is by model transformation, using languages like ATL, a QVT-like Domain Specific Language.

Model-based testing inherits the complexity of the domain or, more particularly, of the related domain models.

Deploying model-based testing

There are various known ways to deploy model-based testing, which include **online testing**, **offline generation of executable tests**, and **offline generation of manually deployable tests**.^[1]

Online testing means that a model-based testing tool connects “directly” to a system under test and tests it dynamically.

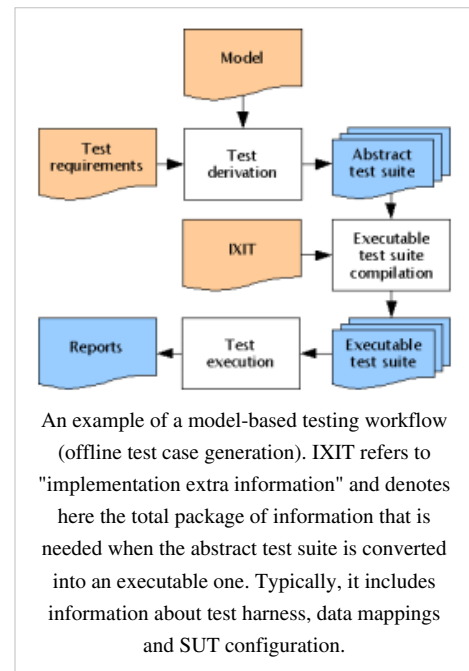
Offline generation of executable tests means that a model-based testing tool generates test cases as a computer-readable asset that can be later deployed automatically. This asset can be, for instance, a collection of Python classes that embodies the generated testing logic.

Offline generation of manually deployable tests means that a model-based testing tool generates test cases as a human-readable asset that can be later deployed manually. This asset can be, for instance, a PDF document in English that describes the generated test steps.

Deriving tests algorithmically

The effectiveness of model-based testing is primarily due to the potential for automation it offers. If the model is machine-readable and formal to the extent that it has a well-defined behavioral interpretation, test cases can in principle be derived mechanically.

Often the model is translated to or interpreted as a finite state automaton or a state transition system. This automaton represents the possible configurations of the system under test. To find test cases, the automaton is searched for executable paths. A possible execution path can serve as a test case. This method works if the model is deterministic or can be transformed into a deterministic one. Valuable off-nominal test cases may be obtained by leveraging un-specified transitions in these models.



Depending on the complexity of the system under test and the corresponding model the number of paths can be very large, because of the huge amount of possible configurations of the system. For finding appropriate test cases, i.e. paths that refer to a certain requirement to proof, the search of the paths has to be guided. For test case generation, multiple techniques have been applied and are surveyed in ^[2].

Test case generation by theorem proving

Theorem proving has been originally used for automated proving of logical formulas. For model-based testing approaches the system is modeled by a set of logical expressions (predicates) specifying the system's behavior. For selecting test cases the model is partitioned into equivalence classes over the valid interpretation of the set of the logical expressions describing the system under development. Each class is representing a certain system behavior and can therefore serve as a test case. The simplest partitioning is done by the disjunctive normal form approach. The logical expressions describing the system's behavior are transformed into the disjunctive normal form.

Test case generation by constraint logic programming and symbolic execution

Constraint programming can be used to select test cases satisfying specific constraints by solving a set of constraints over a set of variables. The system is described by the means of constraints^[3]. Solving the set of constraints can be done by Boolean solvers (e.g. SAT-solvers based on the Boolean satisfiability problem) or by numerical analysis, like the Gaussian elimination. A solution found by solving the set of constraints formulas can serve as a test cases for the corresponding system.

Constraint programming can be combined with symbolic execution. In this approach a system model is executed symbolically, i.e. collecting data constraints over different control paths, and then using the constraint programming method for solving the constraints and producing test cases.

Test case generation by model checking

Model checkers can also be used for test case generation^[4]. Originally model checking was developed as a technique to check if a property of a specification is valid in a model. When used for testing, a model of the system under test, and a property to test is provided to the model checker. Within the procedure of proofing, if this property is valid in the model, the model checker detects witnesses and counterexamples. A witness is a path, where the property is satisfied, whereas a counterexample is a path in the execution of the model, where the property is violated. These paths can again be used as test cases.

Test case generation by using an event-flow model

A popular model that has recently been used extensively for testing software with a graphical user-interface (GUI) front-end is called the event-flow model that represents events and event interactions. In much the same way as a control-flow model represents all possible execution paths in a program, and a data-flow model represents all possible definitions and uses of a memory location, the event-flow model represents all possible sequences of events that can be executed on the GUI. More specifically, a GUI is decomposed into a hierarchy of modal dialogs; this hierarchy is represented as an integration tree; each modal dialog is represented as an event-flow graph that shows all possible event execution paths in the dialog; individual events are represented using their preconditions and effects. An overview of the event-flow model with associated algorithms to semi-automatically reverse engineer the model from an executing GUI software is presented in *this 2007 paper* ^{[5] [6]}. Because the event-flow model is not tied to a specific aspect of the GUI testing process, it may be used to perform a wide variety of testing tasks by defining specialized model-based techniques called event-space exploration strategies (ESES). These ESES use the event-flow model in a number of ways to develop an end-to-end GUI testing process, namely by checking the model, test-case generation, and test oracle creation. Please see the GUI Testing page for more details.

Test case generation by using a Markov chains model

Markov chains are an efficient way to handle Model-based Testing. Test model realized with Markov chains model can be understood as a usage model: we spoke of Usage/Statistical Model Based Testing. Usage models, so Markov chains, are mainly constructed by 2 artifacts : the Finite State Machine (FSM) which represents all possible usage scenario of the system and the Operational Profiles (OP) which qualify the FSM to represent how the system will statically will be used. The first (FSM) helps to know what can be or has been tested and the second (OP) helps to derive operational test cases. Usage/Statistical Model-based Testing starts from the facts that is not possible to exhaustively test a system and that failure can appear with a very low rate.^[7] . This approach offers a pragmatic way to statically derive test cases focused on: improving as prompt as possible the system under test reliability. The company ALL4TEC provides an implementation of this approach with the tool MaTeLo (Markov Test Logic). MaTeLo allows to model the test with Markov chains, derive executables test cases w.r.t the usage testing approach, and assess the system under test reliability with the help of the so called Test Campaign Analysis module.

References

- [1] *Practical Model-Based Testing: A Tools Approach* (<http://www.cs.waikato.ac.nz/~marku/mbt>), Mark Utting and Bruno Legeard, ISBN 978-0-12-372501-1, Morgan-Kaufmann 2007
- [2] John Rushby. Automated Test Generation and Verified Software. Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13. pp. 161-172, Springer-Verlag
- [3] Jefferson Offutt. Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, 17:900-910, 1991
- [4] Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. Software Testing, Verification and Reliability, 19(3):215–261, 2009. URL: <http://www3.interscience.wiley.com/journal/121560421/abstract>
- [5] <http://www.cs.umd.edu/~atif/papers/MemonSTVR2007-abstract.html>
- [6] Atif M. Memon. An event-flow model of GUI-based applications for testing Software Testing, Verification and Reliability, vol. 17, no. 3, 2007, pp. 137-157, John Wiley and Sons Ltd. URL: <http://www.cs.umd.edu/~atif/papers/MemonSTVR2007.pdf>
- [7] Helene Le Guen. Validation d'un logiciel par le test statistique d'usage : de la modelisation de la decision à la livraison, 2005. URL:<ftp://ftp.irisa.fr/techreports/theses/2005/leguen.pdf>

Further reading

- OMG UML 2 Testing Profile; (<http://www.omg.org/cgi-bin/doc?formal/05-07-07.pdf>)
- Eckard Bringmann, Andreas Krämer; Model-based Testing of Automotive Systems (http://www.piketec.com/downloads/papers/Kraemer2008-Model_based_testing_of_automotive_systems.pdf) In: ICST, pp. 485-493, 2008 International Conference on Software Testing, Verification, and Validation, 2008.
- *Practical Model-Based Testing: A Tools Approach* (<http://www.cs.waikato.ac.nz/~marku/mbt>), Mark Utting and Bruno Legeard, ISBN 978-0-12-372501-1, Morgan-Kaufmann 2007.
- *Model-Based Software Testing and Analysis with C#* (<http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=9780521687614>), Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte, ISBN 978-0-521-68761-4, Cambridge University Press 2008.
- *Model-Based Testing of Reactive Systems* (<http://www.springer.com/west/home/computer/programming?SGWID=4-40007-22-52081580-detailsPage=ppmmmedialaboutThisBook>) Advanced Lecture Series, LNCS 3472, Springer-Verlag, 2005.
- Hong Zhu et al. (2008). *AST '08: Proceedings of the 3rd International Workshop on Automation of Software Test* (<http://portal.acm.org/citation.cfm?id=1370042#>). ACM Press. ISBN 978-1-60558-030-2.
- *Requirements for information systems model-based testing* (<http://portal.acm.org/citation.cfm?id=1244306&coll=ACM&dl=ACM&CFID=37894597&CFTOKEN=57312761>)
- *Model-Based Testing Adds Value* (<http://www.methodsandtools.com/archive/archive.php?id=102>), Ewald Roodenrijs, Methods & Tools, Spring 2010.
- *A Systematic Review of Model Based Testing Tool Support* (http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-10-04.pdf), Muhammad Shafique, Yvan Labiche, Carleton University, Technical Report, May 2010.

- *Model-Based Testing for Embedded Systems (Computational Analysis, Synthesis, and Design of Dynamic Systems)* (<http://www.amazon.com/Model-Based-Embedded-Computational-Analysis-Synthesis/dp/1439818452>), Justyna Zander, Ina Schieferdecker, Pieter J. Mosterman, 592 pages, CRC Press, ISBN-10: 1439818452, September 15, 2011.

Modified Condition/Decision Coverage

Modified Condition/Decision Coverage (MC/DC), is used in the standard DO-178B to ensure that Level A software is tested adequately.

To satisfy the MC/DC coverage criterion, during testing all of the below must be true at least once^[1]:

- Each decision tries every possible outcome
- Each condition in a decision takes on every possible outcome
- Each entry and exit point is invoked
- Each condition in a decision is shown to independently affect the outcome of the decision.

Independence of a condition is shown by proving that only one condition changes at a time.

The most critical (Level A) software, which is defined as that which could prevent continued safe flight and landing of an aircraft, must satisfy a level of coverage called *Modified Condition/Decision Coverage* (MC/DC).

Definitions

Condition

A condition is a leaf-level Boolean expression (it cannot be broken down into a simpler Boolean expression).

Decision

A Boolean expression composed of conditions and zero or more Boolean operators. A decision without a Boolean operator is a condition.

Condition Coverage

Every condition in a decision in the program has taken all possible outcomes at least once.

Decision Coverage

Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken all possible outcomes at least once.

Condition/Decision Coverage

Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once.

Modified Condition/Decision Coverage

Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision's outcome independently by varying just that condition while holding fixed all other possible conditions. The condition/decision criterion does not guarantee the coverage of all conditions in the module because in many test cases, some conditions of a decision are masked by the other conditions. Using the modified condition/decision criterion, each condition must be shown to be able to act on the decision outcome by itself, everything else being held fixed. The MC/DC criterion is thus much stronger than the condition/decision coverage.

External links

- What is a "Decision" in Application of Modified Condition/Decision Coverage (MC/DC) and Decision Coverage (DC)? ^[2]

References

- [1] Hayhurst, Kelly; Veerhusen, Dan; Chilenski, John; Rierison, Leanna (May 2001). "A Practical Tutorial on Modified Condition/ Decision Coverage" (<http://shemesh.larc.nasa.gov/fm/papers/Hayhurst-2001-tm210876-MCDC.pdf>). NASA. .
- [2] http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf

Modularity-driven testing

Modularity-driven testing is a term used in the testing of software.

Test Script Modularity Framework

The test script modularity framework requires the creation of small, independent scripts that represent modules, sections, and functions of the application-under-test. These small scripts are then used in a hierarchical fashion to construct larger tests, realizing a particular test case.

Of all the frameworks, this one should be the simplest to grasp and master. It is a well-known programming strategy to build an abstraction layer in front of a component to hide the component from the rest of the application. This insulates the application from modifications in the component and provides modularity in the application design. The test script modularity framework applies this principle of abstraction or encapsulation in order to improve the maintainability and scalability of automated test suites.

Monkey test

In computer science, a **monkey test** is a unit test that runs with no specific test in mind. The monkey in this case is the producer of any input. For example, a monkey test can enter random strings into text boxes to ensure handling of all possible user input or provide garbage files to check for loading routines that have blind faith in their data. The test monkey is technically known to conduct stochastic testing, which is in the category of black-box testing.

The name 'monkey' comes from the adage that 'a thousand monkeys at a thousand typewriters will eventually type out the entire works of Shakespeare'. Cf. the Infinite monkey theorem.

Types of monkey test

Smart Monkey Testing

Input are generated from probability distributions that reflect actual expected usage statistics -- e.g., from user profiles. There are different levels of IQ in smart monkey testing. In the simplest, each input is considered independent of the other inputs. That is, a given test requires an input vector with five components. In low IQ testing, these would be generated independently. In high IQ monkey testing, the correlation (e.g., the covariance) between these input distribution is taken into account. In all branches of smart monkey testing, the input is considered as a single event.^[1]

Brilliant Monkey Testing

The inputs are created from a stochastic regular expression or stochastic finite-state machine model of user behavior. That is, not only are the values determined by probability distributions, but the sequence of values and the sequence of states in which the input provider goes is driven by specified probabilities.^[2]

Dumb Monkey Testing

Inputs are generated from a uniform probability distribution without regard to the actual usage statistics.^[3]

External links

- Unit Testing with Monkeys^[4]
- Using Monkey Test Tools^[5]
- Monkey Tester Software Tool^[6]

References

[1] Visual Test 6 Bible by Thomas R. Arnold, 1998

[2] Visual Test 6 Bible by Thomas R. Arnold, 1998

[3] Visual Test 6 Bible by Thomas R. Arnold, 1998

[4] http://www.mit.jyu.fi/ji/geek/kalle1_monkey.pdf

[5] http://www.stickyminds.com/sitewide.asp?Function=FEATUREDCOLUMN&ObjectId=5054&ObjectType=ARTCOL&btntopic=artcol&tt=LIMITCAT_5054_**WHERE**&tth=H

[6] <http://www.poderico.it/guitester/index.html>

Month of bugs

Month of Bugs is an increasingly popular strategy used by security researchers to draw attention to the lax security procedures of commercial software corporations. The tenet is these corporations have shown themselves to be unresponsive and uncooperative to security alerts and that "responsible disclosure" isn't working properly where they're concerned. To that effect, researchers start a Month of Bugs project for a certain software product and disclose one security vulnerability each day for one month.

The original "Month of Bugs" was the *Month of Browser Bugs* (MoBB) run by security researcher HD Moore.^[1] Subsequent projects include the *Month of Kernel Bugs* (MoKB) which published kernel bugs for Mac OS X, Linux, FreeBSD, Solaris and Windows, as well as four wireless driver bugs;^{[2] [3] [4]} the *Month of Apple Bugs* (MoAB) conducted by researchers Kevin Finisterre and LMH which published bugs related to OS X;^{[5] [6] [7]} and the *Month of PHP Bugs* sponsored by the Hardened PHP team which published 44 PHP bugs.^{[8] [9] [10]}

References

- [1] Kerner, Sean Michael (5 July 2006). "The Month of The Browser Bugs Begins" (<http://www.internetnews.com/security/article.php/3618126>). *InternetNews.com*. QuinStreet Inc.. Retrieved 22 October 2010.
- [2] Mogull, Rich (6 November 2006). "Learn from 'Month of Kernel Bugs'" (http://www.gartner.com/DisplayDocument?doc_cd=144700&ref=g_homelink). *Gartner archive*. Gartner Inc.. Retrieved 22 October 2010.
- [3] Naraine, Ryan (1 November 2006). "Month of Kernel Bugs Launches with Apple Wi-Fi Exploit" (<http://www.eweek.com/c/a/Security/Month-of-Kernel-Bugs-Launches-with-Apple-Wi-Fi-Exploit/>). *eWeek*. Ziff Davis Enterprise Holdings Inc.. Retrieved 22 October 2010.
- [4] Evers, Joris (2 November 2006). "Apple wireless flaw revealed" (<http://www.zdnet.co.uk/news/security-threats/2006/11/02/apple-wireless-flaw-revealed-39284508/>). *ZDNet*. CBS Interactive.. Retrieved 22 October 2010.
- [5] McMillan, Robert (20 December 2006). "Apple Bug-Hunt Begins" (http://www.pcworld.com/article/128282/apple_bughunt_begins.html). *PC World*. PCWorld Communications, Inc.. Retrieved 22 October 2010.
- [6] Leyden, John (20 December 2006). "Month of Apple bugs planned for January" (http://www.theregister.co.uk/2006/12/20/month_of_apple_bugs/). *The Register*. The Register.. Retrieved 22 October 2010.
- [7] Naraine, Ryan (19 December 2006). "Coming in January: Month of Apple Bugs" (http://securitywatch.eweek.com/apple/coming_in_january_month_of_apple_bugs.html). *eWeek Security Watch*. Ziff Davis Enterprise Holdings Inc.. Retrieved 22 October 2010.
- [8] Prince, Brian (3 March 2007). "Month of PHP Bugs Begins" (<http://www.eweek.com/c/a/Security/Month-of-PHP-Bugs-Begins/>). *eWeek*. Ziff Davis Enterprise Holdings Inc.. Retrieved 22 October 2010.
- [9] Naraine, Ryan (1 March 2007). "Flaw trifecta kicks off Month of PHP bugs" (<http://www.zdnet.com/blog/security/flaw-trifecta-kicks-off-month-of-php-bugs/107>). *ZDNet*. CBS Interactive.. Retrieved 22 October 2007.
- [10] Naraine, Ryan (4 May 2007). "Controversial 'month of bugs' getting security results" (http://www.zdnet.com/blog/security/controversial-month-of-bugs-getting-security-results/189?tag=mantle_skin;content). *ZDNet*. CBS Interactive.. Retrieved 22 October 2010.

Further reading

- McMillan, Robert (17 March 2007). "Hackers Promise Month of MySpace Bugs" (http://www.pcworld.com/article/129933/hackers_promise_month_of_myspace_bugs.html). *PC World*. PCWorld Communications, Inc.. Retrieved 22 October 2010.

External links

- Month of Kernel Bugs (MoKB) archive (<http://projects.info-pull.com/mokb/>)
- Kernel Fun (<http://kernelfun.blogspot.com/>): *Month of the Kernel Bugs* blog
- Month of Apple Bugs (MoAB) archive (<http://projects.info-pull.com/moab/>)
- Apple Fun (<http://applefun.blogspot.com/>): *Month of the Apple Bugs* blog
- Info-pull.com blog (<http://blog.info-pull.com/>): A complementary blog from the hosts of *MoKB* and *MoAB*
- the Month of PHP Security (<http://php-security.org/>)

Mutation testing

For the biological term, see: Gene mutation analysis.

Mutation testing (or *Mutation analysis* or *Program mutation*) is a method of software testing, which involves modifying programs' source code or byte code in small ways.^[1] A test suite that does not detect and reject the mutated code is considered defective. These so-called *mutations*, are based on well-defined *mutation operators* that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Aim

Tests can be created to verify the correctness of the implementation of a given software system, but the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. (This technological problem is itself an instance of a deeper philosophical problem named "Quis custodiet ipsos custodes?" ["Who will guard the guards?"].) In this context, mutation testing was pioneered in the 1970s to locate and expose weaknesses in test suites. The theory was that if a mutation was introduced without the behavior (generally output) of the program being affected, this indicated either that the code that had been mutated was never executed (redundant code) or that the testing suite was unable to locate the injected fault. In order for this to function at any scale, a large number of mutations had to be introduced into a large program, leading to the compilation and execution of an extremely large number of copies of the program. This problem of the expense of mutation testing had reduced its practical use as a method of software testing, but the increased use of object oriented programming languages and unit testing frameworks has led to the creation of mutation testing tools for many programming languages as a means to test individual portions of an application.

Historical overview

Mutation testing was originally proposed by Richard Lipton as a student in 1971,^[2] and first developed and published by DeMillo, Lipton and Sayward. The first implementation of a mutation testing tool was by Timothy Budd as part of his PhD work (titled *Mutation Analysis*) in 1980 from Yale University.

Recently, with the availability of massive computing power, there has been a resurgence of mutation analysis within the computer science community, and work has been done to define methods of applying mutation testing to object oriented programming languages and non-procedural languages such as XML, SMV, and finite state machines.

In 2004 a company called Certess Inc. extended many of the principles into the hardware verification domain. Whereas mutation analysis only expects to detect a difference in the output produced, Certess extends this by verifying that a checker in the testbench will actually detect the difference. This extension means that all three stages of verification, namely: activation, propagation and detection are evaluated. They have called this functional qualification.

Fuzzing is a special area of mutation testing. In fuzzing, the messages or data exchanged inside communication interfaces (both inside and between software instances) are mutated, in order to catch failures or differences in processing the data. Codenomicon^[3] (2001) and Mu Dynamics (2005) evolved fuzzing concepts to a fully stateful mutation testing platform, complete with monitors for thoroughly exercising protocol implementations.

Mutation testing overview

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a *mutant*. If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be *killed*.

For example, consider the following C++ code fragment:

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

The condition mutation operator would replace `&&` with `||` and produce the following mutant:

```
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

Now, for the test to kill this mutant, the following condition should be met:

- Test input data should cause different program states for the mutant and the original program. For example, a test with `a = 1` and `b = 0` would do this.
- The value of 'c' should be propagated to the program's output and checked by the test.

Weak mutation testing (or *weak mutation coverage*) requires that only the first condition is satisfied. *Strong mutation testing* requires that both conditions are satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.

Equivalent mutants

Many mutation operators can produce equivalent mutants. For example, consider the following code fragment:

```
int index = 0;

while (...)
{
    ...;
    index++;

    if (index == 10) {
        break;
    }
}
```

Boolean relation mutation operator will replace `==` with `>=` and produce the following mutant:

```
int index = 0;
```

```
while (...)
{
    ...;
    index++;

    if (index >= 10) {
        break;
    }
}
```

However, it is not possible to find a test case that could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called *equivalent mutants*.

Equivalent mutants detection is one of biggest obstacles for practical usage of mutation testing. The effort needed to check if mutants are equivalent or not, can be very high even for small programs.^[4]

Mutation operators

A variety of mutation operators were explored by researchers. Here are some examples of mutation operators for imperative languages:

- Statement deletion.
- Replace each boolean subexpression with *true* and *false*.
- Replace each arithmetic operation with another one, e.g. + with *, - and /.
- Replace each boolean relation with another one, e.g. > with >=, == and <=.
- Replace each variable with another variable declared in the same scope (variable types should be the same).

These mutation operators are also called traditional mutation operators. Beside this, there are mutation operators for object-oriented languages^[5], for concurrent constructions^[6], complex objects like containers^[7] etc. They are called class-level mutation operators. For example the MuJava tool offers various class-level mutation operators such as: Access Modifier Change, Type Cast Operator Insertion, Type Cast Operator Deletion. Moreover, mutation operators have been developed to perform security vulnerability testing of programs^[8]

References

- [1] A Practical System for Mutation Testing: Help for the Common Programmer (<http://cs.gmu.edu/~offutt/rsrch/papers/practical.pdf>) by A. Jefferson Offutt.
- [2] Mutation 2000: Uniting the Orthogonal (<http://cs.gmu.edu/~offutt/rsrch/papers/mut00.pdf>) by A. Jefferson Offutt and Roland H. Untch.
- [3] Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. 2001. (<http://www.codenomicon.com/resources/publications.shtml>)
- [4] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38:235–253, 1997.
- [5] MuJava: An Automated Class Mutation System (<http://www.cs.gmu.edu/~offutt/rsrch/papers/mujava.pdf>) by Yu-Seung Ma, Jeff Offutt and Yong Rae Kwo.
- [6] Mutation Operators for Concurrent Java (J2SE 5.0) (http://www.irisa.fr/manifestations/2006/Mutation2006/papers/14_Final_version.pdf) by Jeremy S. Bradbury, James R. Cordy, Juergen Dingel.
- [7] Mutation of Java Objects (<http://www.cs.colostate.edu/~bieman/Pubs/AlexanderBiemanGhoshJiISSRE02.pdf>) by Roger T. Alexander, James M. Bieman, Sudipto Ghosh, Bixia Ji.
- [8] Mutation-based Testing of Buffer Overflows, SQL Injections, and Format String Bugs (<http://qspace.library.queensu.ca/handle/1974/1359>) by H. Shahriar and M. Zulkernine.

Further reading

- Aristides Dasso, Ana Funes (2007). *Verification, Validation and Testing in Software Engineering*. Idea Group Inc. ISBN 1591408512. See Ch. VII *Test-Case Mutation* for overview on mutation testing.
- Paul Ammann, Jeff Offutt (2008). *Introduction to Software Testing*. Cambridge University Press. ISBN 0-52188-038-1. See Ch. V *Syntax Testing* for an overview of mutation testing.
- Yue Jia, Mark Harman (September 2009). "An Analysis and Survey of the Development of Mutation Testing" (<http://www.dcs.kcl.ac.uk/pg/jiayue/repository/TR-09-06.pdf>) (PDF). *CREST Centre, King's College London, Technical Report TR-09-06*.

External links

- Mutation testing (<http://cs.gmu.edu/~offutt/rsrch/mut.html>) list of tools and publications by Jeff Offutt.
- Mutation Testing Repository (<http://www.dcs.kcl.ac.uk/pg/jiayue/repository/>) A publication repository that aims to provide a full coverage of the publications in the literature on Mutation Testing.
- Jumble (<http://jumble.sourceforge.net/>) Bytecode based mutation testing tool for Java
- PIT (<http://pitest.org/>) Bytecode based mutation testing tool for Java
- Jester (<http://jester.sourceforge.net/>) Source based mutation testing tool for Java
- Heckle (<http://glu.ttono.us/articles/2006/12/19/tormenting-your-tests-with-heckle>) Mutation testing tool for Ruby
- Nester (<http://nester.sourceforge.net/>) Mutation testing tool for C#
- Mutagenesis (<https://github.com/padraic/mutagenesis>) Mutation testing tool for PHP

National Software Testing Laboratories

National Software Testing Laboratories, or **NSTL**, is an American company, established in 1983, which tests computer hardware and software. The company provides certification (such as WHQL and Microsoft Windows Mobile certification), quality assurance, and benchmarking services. NSTL is the only authorized testing organization for the Qualcomm TRUE BREW platform.

External links

- Company website ^[1]

References

[1] <http://www.nstl.com/>

NMock

NMock is a library of mock objects to be used in .net development for Test Driven Development. NMock is an open source project that was inspired by the JMock project. The current version as of 2011 is NMock 3.

External links

- <http://nmock3.codeplex.com> ^[1]
- <http://www.nmock.org> ^[2]

References

[1] <http://nmock3.codeplex.com>

[2] <http://www.nmock.org>

Non-functional testing

Non-functional testing is the testing of a software application for its non-functional requirements. The names of many non-functional tests are often used interchangeably because of the overlap in scope between various non-functional requirements. For example, software performance is a broad term that includes many specific requirements like reliability and scalability.

Non-functional testing includes:

- Baseline testing
 - Compatibility testing
 - Compliance testing
 - Documentation testing
 - Endurance testing
 - Load testing
 - Localization testing and Internationalization testing
 - Performance testing
 - Recovery testing
 - Resilience testing
 - Security testing
 - Scalability testing
 - Stress testing
 - Usability testing
 - Volume testing
-

Non-Regression testing

Non-Regression testing (NRT) is an approach of software testing which verifies if the desired previous functionalities of a software application were not compromised after the introduction of new procedures or functional modifications in a new release. The observation of undesired behaviours generally indicates a software regression, i.e. the new features added to the software introduced software bugs in the functionalities that were working in the right way in its previous version.

Introduction: Development of a Software

The development process of a software application can be divided in several steps, where each one culminates in a new software version containing a certain number of new features until the last release, when all the contents that satisfy the customer's requirements are available. As the complexity of software architecture grows up, step by step, higher is the probability of occurrence of bugs during this development process. Bugs can occur when the software code is modified mainly for two reasons:

- new procedure in conflict with an old one;
- modification to upgrade a pre-existing procedure.

Usually, the occurrence of a software bug can result in an unexpected delay to the project. Due to time-to-market restrictions, the validation phase of software functionalities must be well organized and efficient so that it lasts as short as possible. In this context, Non-Regression testing provides a systematic procedure to have a fast and efficient validation phase and discover as soon as possible the software bugs hidden in the architecture.

How to perform a Non-Regression Test

A Non-Regression test can be performed according the following steps:

1. Define a benchmark software release;
2. Define a set of routines able to stimulate as many functionalities as possible of the software;
3. Launch these routines on both software (the benchmark and the new release under test) and acquire data which represents their behaviour;
4. Analyse this data with a post-processing tool, able to provide statistic results;
5. Report the outcome.

Exploratory testing is performed following similar steps, but it differs from NRT concerning their analysis phase and focus, searching, hence, for different results and conclusions. The NRT aims to check if any undesired behaviour comes out after the last modifications applied to the software. There, the new behaviour of the application is previously known, making possible the identification of an eventual regression (bug). Exploratory testing, on the other hand, seeks to find out how the software actually works, conciliating simultaneous testing and learning, and stimulating testers to create new test cases.^[1]

Regression and Non-Regression Testing

While the intent of regression testing is to assure that a bug fix has been successfully corrected, based on the error that was found, by retesting the modified software, the Non-Regression Testing aims to verify the no introduction of software bugs due to the introduction of new features.^[2] In general, this difference of definitions can be assumed as based on the outcome of each test, as follows.

When a new software version is released without any new feature regarding to the previous one (i.e. the differences between them are restricted to bug fixes or software optimization), both the releases are expected to present the same functionalities. In this case, the tests applied to both versions are not expected to result in different behaviours, but only assure the fixing and/or no introduction of new bugs. This testing methodology characterizes the regression testing.

On the other hand, when the new release presents new functionalities or improvements that lead the software to behaviour in a different way, then the tests performed on the previous and new version can result in:

- desired differences, related to an expected new behaviour; and
- undesired differences, which indicate a software regression generally caused by a side-effect bug.

In this case we can talk about Non-Regression Testing.

Who performs the Non-Regression Testing

Once the customer has set all the requirements, the supplier will introduce all the contents, release by release, until the final one. In this context, NRT can be performed by both the customer and the supplier.

It can be made by the supplier as a beta testing service to guarantee a higher quality product with a very low percentage probability of bugs. Basically the client is equipped with simulation environment that enables an easy way to perform routines and acquire data. In case of regression the supplier, owing the know-how, can quickly solve the problem and avoid releasing a malfunction software version to the customer.

On the other hand, NRT can be performed by the customer as an acceptance testing in order to prevent his final product from damages and eventually charge the supplier for the mismatch with requirements. Moreover, the customer, having a reduced know-how about the software structure, can perform the NRT as a black-box testing and, after meeting a regression, refuse the new software release.

NRT Automotive Applications

Throughout the years Engine Control Unit (ECU) software requirements are getting more complex and harder to reach due to the more and more stringent emission norms and ambitious performance in terms of fuel consumption and power request, which also increase the demand and complexity of in-vehicle driving tests and diagnostic functionalities. As a consequence, along engine control systems development, each new software release results from a sequence of many others, each one introducing new functions seeking to satisfy, time after time, the crescent demands. In this context, Non-Regression testing is useful to verify that the performance and robustness of each software release does not decrease in relation with the previous one, or, in other terms, does not introduce regression.

NRT is applied, along each software release testing phase, as the final stage during integration testing, right before the execution of system testing and after the module testing (or unit testing) phase.^[3] In the module testing phase, single software modules are evaluated individually, which allows the identification of elementary errors like overflow, underflow, round-off, and also discrepancies between algorithm model simulation results and the signals coming from the Engine Management System (EMS). The integration testing phase, performed afterwards, aims to verify if the tested module is correctly integrated in the overall software system. Finally, functional testing (also called validation testing) is applied to validate the algorithms concerning functional requirements. This stage is usually performed after the calibration phase and characterizes an overall system testing, concluding the new

software testing phase, and allowing, therefore, its release.

In automotive applications, non-regression testing is performed as follows:

1. Selection of test manoeuvres and definition of engine parameters to be monitored;
2. Execution of the selected manoeuvres on benchmark software and the software under test;
3. Post-processing and analysis of data acquired during these tests.

The selected test manoeuvres must be able to stimulate as many algorithms implemented in the software as possible. Cold start, Overshoot of Rounds per minute, and ECE cycle (a standard manoeuvre used to calibrate On-Board diagnosis) are relevant examples. In addition, the engine parameters selected to be monitored must represent the engine global operating state along the manoeuvres executed, such as Accelerator pedal deflection, Engine speed, Vehicle speed, Engine temperature and Throttle body opening percentage. It's also necessary to monitor the mean variables of the control chain of Air and Torque estimation. All these diagnostic variables must be kept under control during the execution of the manoeuvres.

The tests are performed in simulation environments such as Hardware-in-the-loop (HIL) simulators or Micro HIL (feed-forward systems that work as HIL downsized simulators), which support the drawing and execution of complex maneuvers usually very difficult to perform on a real engine or car (mainly because of time, cost and equipment restrictions).^{[3] [4]}

Afterwards, a post-processing tool is required to process the acquired data, offering graphic analysis and statistical data, generally addressed to skilled personal able to identify possible regression on the software. This kind of tool can also be endowed with an automatic report generator, which gathers in a single document all the analysis results and conclusions from the comparison between the two software releases during the NRT.

References

- [1] Kaner, Cem; Bach, James; Pettichord, Bret (2001). *Lessons Learned in Software Testing*. John Wiley & Sons. ISBN 0471081124.
- [2] T. Jérón, J.M. Jézéquel, Y. Le Traon and P. Morel, "Efficient Strategies for Integration and Regression Testing of OO Systems", *10th IEEE International Symposium on Software Reliability Engineering (ISSRE'99)*, Florida, p. 260-269, November 1999. By VerTeCs (<http://www.irisa.fr/vertecs/Publis//Publi/Jeron-Jezequel-LeTraon-Morel-ISSRE99.html#download>) (accessed on 01/10/10).
- [3] A. Palladino, G. Fiengo, F. Giovagnini, and D. Lanzo, "A Micro Hardware-In-the-Loop Test System", *IEEE European Control Conference*, 2009. Abstract (<https://controls.papercept.net/conferences/scripts/abstract.pl?ConfID=5&Number=940>) (accessed on 01/10/10).
- [4] S.Raman, N. Sivashankar, W. Milam, W. Stuart, and S. Nabi, "Design and Implementation of HIL Simulators for Powertrain Control System Software Development", *Proceedings of the American Control Conference*, 1999.

External links

- User Guide for Non-Regression Tests (http://hydrolab.irisa.fr/doc/developing/non_regression_tests.htm)
- Advanced Testing Methods for Automotive Software (http://www.win.tue.nl/~mvdbrand/courses/sse/0809/papers/mentorpaper_28585.pdf)

Operational Acceptance Testing

Operational Acceptance Testing (OAT) is a type of software testing, used mainly in software support and software maintenance projects. This type of testing focuses on the operational readiness of the system to be supported, or which is to become the production environment. Hence, it is also known as **operational readiness testing**. Functional testing of applications is not included or merged in OAT.

It may include checking the backup facilities, maintenance and disaster recovery procedures. In OAT changes are made to environmental parameters which the application uses to run smoothly. For example, with Microsoft Windows applications with a mixed or hybrid architecture, this may include: Windows services, configuration files, web services, XML files, COM+ components, web services, IIS, stored procedures in databases, etc. This type of testing is conducted before user acceptance testing.

The approach used in OAT includes these steps:

- Build the system,
- Deploy the application,
- Supportability of the system.
- Validate the backup procedure setup for the system

Then checks are made on how the application is behaving, and moreover how the system is behaving as a whole in these conditions. Backup procedures are also checked to ensure proper operation under emergency conditions.

For running the OAT test cases, the tester normally has exclusive access to the system or environment. This means that a single tester would be executing the test cases at a single point of time. For OAT the exact OR quality gates are defined, both Entry and Exit Gate. All activities are listed which would be part and covered in the different phases of testing, with primary emphasis be on the operational part of the system.

There are various aspects of OAT -

- Fail over testing (Resilience testing)
 - Component fail over
 - Network fail over
- Alerting / Alarming (to ensure proper alerts are configured in the system if something goes wrong)
- Backup and recovery testing - to ensure that backup and recovery is successful

Oracle (software testing)

An **oracle** is a mechanism used by software testers and software engineers for determining whether a test has passed or failed.^[1] It is used by comparing the output(s) of the system under test, for a given test case input, to the outputs that the oracle determines that product should have. Oracles are always separate from the system under test.^[2]

Common oracles include:

- specifications and documentation,^[3]
- other products (for instance, an oracle for a software program might be a second program that uses a different algorithm to evaluate the same mathematical expression as the product under test)
- an *heuristic oracle* that provides approximate results or exact results for a set of a few test inputs,^[4]
- a *statistical oracle* that uses statistical characteristics,^[5]
- a *consistency oracle* that compares the results of one test execution to another for similarity,^[6]
- a *model-based oracle* that uses the same model to generate and verify system behavior,
- or a human being's judgment (i.e. does the program "seem" to the user to do the correct thing?).^[2]

References


- [1] A Course in Black Box Software Testing (<http://www.testingeducation.org/k04/OracleExamples.htm>), Cem Kaner 2004
- [2] *An Integrated Approach to Software Engineering*, Pankaj Jalote 2005, ISBN 038720881X
- [3] "Generating a test oracle from program documentation", Peters and Parnas, 1994 in *Proceedings of the 1994 International Symposium on Software Testing and Analysis*
- [4] Heuristic Test Oracles (http://www.softwarequalitymethods.com/Papers/STQE_Heuristic.pdf), Douglas Hoffman, Software Testing & Quality Engineering Magazine, 1999
- [5] Test Oracles Using Statistical Methods (<http://www.mathematik.uni-ulm.de/sai/mayer/publications/oracles.pdf>), Johannes Mayer and Ralph Guderlei
- [6] Analysis of a Taxonomy for Test Oracles (<http://www.softwarequalitymethods.com/Papers/OracleTax.pdf>), Douglas Hoffman, Quality Week, 1998

Bibliography

- Binder, Robert V. (1999). "Chapter 18 - Oracles" in "Testing Object-Oriented Systems: Models, Patterns, and Tools". Addison-Wesley Professional, 7 November 1999. ISBN-13: 978-0201809381.

Original Software

Original Software

	
Type	Private
Industry	software testing
Founded	December 1996
Founder(s)	Colin Armitage
Products	Qualify, TestDrive, TestDrive-Assist, TestBench, TestSmart
Website	http://www.origsoft.com/

Original Software is a privately held company providing automatic software testing products and services.

History

Original Software was formed in December 1996 and started trading in May 1997. It was founded by Colin Armitage.

In the initial years, Original Software focused on IBM iSeries platform.

In 2007 a manual testing solution was introduced.

In 2010 it was listed as one of "Twenty companies to watch in 2010" by CIO UK.^[1]

Products

Original Software's solutions include:^[2]

- Qualify - an Application Quality Management (AQM) solution uniting all aspects of the software development lifecycle
- TestDrive - a test automation tool
- TestDrive-Assist - a tool for dynamic manual testing
- TestBench - a test data management and verification tool
- TestSmart - a tool for automated creation of optimised variable data

References

- [1] MWD and Martin Veitch (12 January 2010). "Twenty companies to watch in 2010 - In-Depth - CIO UK Magazine" (<http://www.cio.co.uk/article/3210025/twenty-companies--to-watch-in-2010/>). *cio.co.uk*. Retrieved 15 September 2010.
- [2] "Software Testing - Software Application Quality Solutions" (<http://www.origsoft.com/products/>). *origsoft.com*. Retrieved 3 August 2010.
- Thomas E. Murphy (31 July 2009). "Magic Quadrant for Integrated Software Quality Suites" (http://www.gartner.com/DisplayDocument?id=1106812&ref=g_fromdoc). Gartner.
 - Bola Rotibi (14 October 2008). "On the Radar: Original Software" (<http://www.mwdadvisors.com/library/detail.php?id=132>). MWD Advisors.
 - Bola Rotibi (3 Mar 2010). "Original Software qualifies its role in Application Quality Management with new Qualify tool" (<http://www.mwdadvisors.com/library/detail.php?id=252>). MWD Advisors.
 - Chandranshu Singh and Michael Azoff (January 2010). "TECHNOLOGY AUDIT Qualify v1.5 Original Software" (http://www.origsoft.com/_assets/client/docs/pdf/analyst-reports/butler_report.pdf). Ovum Butler Group.

External links

- Official website (<http://http://www.origsoft.com/>)

Oulu University Secure Programming Group

The **Oulu University Secure Programming Group (OUSPG)** is a research group at the University of Oulu that studies, evaluates and develops methods of implementing and testing application and system software in order to prevent, discover and eliminate implementation level security vulnerabilities in a pro-active fashion. The focus is on implementation level security issues and software security testing.

History

OUSPG has been active as an independent academic research group in the Computer Engineering Laboratory in the Department of Electrical and Information Engineering in the University of Oulu since summer 1996.

OUSPG is most known for its participation in protocol implementation security testing, which they called Robustness testing, using the PROTOS mini-simulation method.^[1]

The PROTOS was co-operated project with VTT and number of industrial partners. The project developed different approaches of testing implementations of protocols using black-box (i.e. functional) testing methods. The goal was to support pro-active elimination of faults with information security implications, promote awareness in these issues and develop methods to support customer driven evaluation and acceptance testing of implementations. Improving the security robustness of products was attempted through supporting the development process.

The most notable result of the PROTOS project was the result of the c06-snmp test suite, which discovered multiple vulnerabilities in SNMP.

The work done in PROTOS is continued in PROTOS-GENOME, which applies automatic structure inference combined with domain specific reasoning capabilities to enable automated black-box program robustness testing tools without having prior knowledge of the protocol grammar. This work has resulted in a large number of vulnerabilities being found in archive file and anti-virus products.

Commercial spin-offs

The group has produced two spin-off companies, Codenomicon continues the work of the PROTOS and Clarified Networks the work in FRONTIER.

References

[1] Kaksonen, Rauli. 2001. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications 448. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.).

As of 12:21, 30 July 2009 (UTC), this article is derived in whole or in part from University of Oulu. The copyright holder has licensed the content utilized under CC-BY-SA and GFDL. All relevant terms must be followed. The original text was at "Oulu University Secure Programming Group" (<http://www.ee.oulu.fi/research/ouspg/>).

- Kaksonen, Rauli. 2001. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications 447. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.). <http://www.inf.vtt.fi/pdf/publications/2001/P448.pdf>
- <http://www.ee.oulu.fi/research/ouspg/>

External links

- <http://www.securityfocus.com/news/474>
- <https://www.cert.fi/haavoittuvuudet/joint-advisory-archive-formats.html>

Pair Testing

Pair Testing is a software development technique in which two team members work together at one keyboard to test the software application. One does the testing and the other analyzes or reviews the testing. This can be done between one Tester and Developer or Business Analyst or between two testers with both participants taking turns at driving the keyboard.

Description

This can be more related to Pair Programming and Exploratory testing of Agile Software Development where two team members are sitting together to test the software application. This will help both the members to learn more about the application. This will narrow down the root cause of the problem while continuous testing. Developer can find out which portion of the source code is affected by the bug. This track can help to make the solid test cases and narrowing the problem for the next time.

Benefits and Drawbacks

The developer can learn more about the software application by exploring with the tester. The tester can learn more about the software application by exploring with the developer.

Less participation is required for testing and for important bugs root cause can be analyzed very easily. The tester can very easily test the initial bug fixing status with the developer.

This will make the developer to come up with great testing scenarios by their own

This can not be applicable to scripted testing where all the test cases are already written and one has to run the scripts. This will not help in the evolution of any issue and its impact.

Usage

This is more applicable where the requirements and specifications are not very clear, the team is very new, and needs to learn the application behavior quickly.

This follows the same principles of pair programming; the two team members should be in the same level.

Parameter validation

In computer software, the term **parameter validation**^{[1] [2]} is the automated processing, in a module, to validate the spelling or accuracy of parameters passed to that module. The term has been in common use for over 30 years.^[1] Specific best practices have been developed, for decades, to improve the handling of such parameters.^{[1] [2] [3]}

Notes

[1] "Parameter validation for software reliability", G.B. Alleman, 1978 (*see below*: References).

[2] "Parameter Validation for Floats", MSDN.Microsoft.com, 2007, webpage: MSDN-862 (<http://social.msdn.microsoft.com/forums/en-US/sqlreportingservices/thread/9cbc23b8-8709-4053-90c3-bd4818eda862/>).

[3] "Feedback: Attribute-based method parameter validation and error handling", 2007, webpage: VStudio-327 (<http://connect.microsoft.com/VisualStudio/feedback/ViewFeedback.aspx?FeedbackID=97327>).

References

- "Parameter validation for software reliability", G.B. Alleman, 1978, webpage: ACM-517 (<http://portal.acm.org/citation.cfm?id=987517>): paper presents a method for increasing software reliability through parameter validation.

Partial concurrent thinking aloud

Partial Concurrent Thinking Aloud (or **partial concurrent think-aloud**, or **PCTA**) is a method used to gather data in usability testing with screen reader users. It is a particular kind of think aloud protocol (or TAP) created by Stefano Federici and Simone Borsci^[1] at the Interuniversity Center for Research on Cognitive Processing in Natural and Artificial Systems^[2] of University of Rome "La Sapienza". The Partial Concurrent Thinking Aloud is built up in order to create a specific usability assessment technique for blind users, eligible to maintain the advantages of concurrent and retrospective thinking aloud while overcoming their limits. Using PCTA blind users' verbalizations of problems could be more pertinent and comparable to those given by sighted people who use a concurrent protocol. In the usability evaluation with blind people, the retrospective thinking aloud is often adopted as a functional solution to overcome the structural interference due to thinking aloud and hearing the screen reader imposed by the classic thinking aloud technique; such a solution has yet a relapse in the evaluation method, because the concurrent and the retrospective protocols measure usability from different points of view, one mediated by navigation experience (retrospective) one more direct and pertinent (concurrent)^[3]. The use of PCTA could be widened to both summative and formative usability evaluations with mixed panels of users, thus extending the number of problems' verbalizations according to disabled users' divergent navigation processes and problem solving strategies.

Cognitive assumptions of Partial Concurrent Thinking Aloud

In general, in the usability evaluation both retrospective and concurrent TAP could be used according to the aims and goals of the study. Nevertheless, when a usability evaluation is carried out with blind people several studies propose to use the retrospective TAP: indeed, using a screen reader and talking about the way of interacting with the computer implies a structural interference between action and verbalization. Undoubtedly, cognitive studies provided a lot of evidence supporting the idea that individuals can listen, verbalize, or manipulate, and rescue information in multiple task condition. As Colin Cherry^[4] showed, subjects, when listening to two different messages from a single loudspeaker, can separate sounds from background noise, recognize the gender of the speaker, the direction, and the pitch (cocktail party effect). At the same time, subjects that must verbalize the content of a message (attended message) listening to two different message simultaneously (attended and unattended message) have a reduced ability to report the content of the attended message, while they are unable to report the content of the unattended message. Moreover, K. Anders Ericsson and Walter Kintsch^[5] showed that, in a multiple task condition, subjects' ability of rescuing information is not compromised by an interruption of the action flow (as it happens in the concurrent thinking aloud technique), thanks to the "Long Term Working Memory mechanism" of information retrieval (Working Memory section Ericsson and Kintsch). Even if users can listen, recognize, and verbalize multiple messages in a multiple task condition and they can stop and restart actions without losing any information, other cognitive studies underlined that the overlap of activities in a multiple task condition have an effect on the goal achievement: Kemper, Herman and Lian^[6], analysing the users' abilities to verbalize actions in a multiple task condition, showed that the fluency of a user's conversation is influenced by the overlap of actions. Adults are likely to continue to talk as they navigate in a complex physical environment. However, the fluency of their conversation is likely to change: Older adults are likely to speak more slowly than they would if resting; Young adults continue to speak just as rapidly while walking as while resting, but they adopt a further set of speech accommodations, reducing sentence length, grammatical complexity, and propositional density. Just by reducing length, complexity, and propositional density adults free up working memory resources. We do not know how and how much the content of verbalizations could be influenced by the strategy of verbalization (i.e. the modification of fluency and the complexity in a multiple task condition). Anyway, we well know that users in the concurrent thinking aloud verbalize the problems in a more accurate and pertinent way (i.e. more focused on the problems directly perceived during the interaction) than in the retrospective one^[7]^[8]. The pertinence is granted to the user by the proximity of action-verbalization-next action; this multiple task proximity compels the subject to apply a strategy of verbalization

that reduce the overload of the working memory. However, for blind users this time proximity between action and verbalization is lost: the use of the screen reader, in fact, increase the time for verbalization (i.e. in order to verbalize, blind users must first stop the [screen reader] and then restart it).

Protocol of Partial Concurrent Thinking Aloud

PCTA method is composed of two sections, one concurrent and one retrospective:

The first section is a modified concurrent protocol built up according to the three concurrent verbal protocols criteria described by K. Anders Ericsson and Herbert Simon^{[9] [10]}:

The first criterion
Subjects should be talking about the task at hand, not about an unrelated issue. In order to respect this rule, the time between problem retrieval, thinking and verbalization must be minimized to avoid the influence of a long perceptual reworking and the consequent verbalization of unrelated issues. Blind participants, using a screen reader, increase the time latency between identification and verbalization of a problem. To minimize this latency, users are trained to ring a desk-bell that stops both time and navigation. During this suspension, users can create a memory sign (i.e. ring the bell) and restart immediately the navigation. This setting modification allows to avoid the cognitive limitation problem and the influence of perceptual reworking, also creating a memory sign for the retrospective analysis.
The second criterion
To be pertinent, verbalizations should be logically consistent with the verbalizations that just preceded them. For any kind of user it is hard to be pertinent and consistent in a concurrent verbal protocol. Therefore, the practitioners could generally interrupt the navigation and ask for a clarification or stimulate the users to verbalize in a pertinent way. In order to do so and stop navigation to screen reader users, we propose to negotiate a specific physical sign with them: The practitioner, sitting behind the user, will put his hand on the user's shoulder. This physical sign grants the verbalization pertinence and consistence.
The third criterion
A subset of the information needed during the task performance should be remembered. The concurrent model is based on the link between working memory and time latency. The proximity between the occurrence of a thought and its verbal report allows users to verbalize on the basis of their working memory.

The second PCTA section is a retrospective one in which users analyse those problems previously verbalized in a concurrent way. The memory signs, created by users ringing the desk-bell, overcome the limits of classic retrospective analysis; indeed, these signs allow the users to be pertinent and consistent with their concurrent verbalization, thus avoiding the influence of long term memory and perceptual reworking.

References

- [1] Borsci, S., & Federici, S. (2009). "The Partial Concurrent Thinking Aloud: A New Usability Evaluation Technique for Blind Users". In P. L. Emiliani, L. Burzagli, A. Como, F. Gabbanini, & A. L. Salminen. *Assistive technology from adapted equipment to inclusive environments*. 25. IOS Press. pp. 421–425.
- [2] <http://w3.uniroma1.it/econal/>
- [3] Federici, S., Borsci, S., & Stamerra, G. (Accepted November 2009). "Web usability evaluation with screen reader users: implementation of the partial concurrent thinking aloud technique". *Cognitive processing* 11: 263. doi:10.1007/s10339-009-0347-y.
- [4] Cherry, E.C. (1953). "Some experiments on the recognition of speech, with one and with two ears". *Journal of the Acoustical Society of America* 25 (5): 975–979. doi:10.1121/1.1907229.
- [5] Ericsson, K.A., Kintsch, W. (1995). "Long-Term Working Memory". *Psychological Review* 102 (2): 211–245. doi:10.1037/0033-295X.102.2.211. PMID 7740089.
- [6] Kemper, S., Herman, R.E., & Lian, C.H.T. (2003). "The Costs of Doing Two Things at Once for Young and Older Adults: Talking While Walking, Finger Tapping, and Ignoring Speech or Noise". *Psychology and Aging* 18 (2): 181–192. doi:10.1037/0882-7974.18.2.181. PMID 12825768.
- [7] Bowers, V.A & Snyder, H.L. (2003). *Concurrent versus retrospective verbal protocols for comparing window usability*. Human Factors Society 34th Meeting, 8-12 October 1990 HFES, Santa Monica. pp. 1270–1274.
- [8] Van den Haak, M.J. & De Jong, M.D.T. (2003). *Exploring Two Methods of Usability Testing: Concurrent versus Retrospective Think-Aloud Protocols*. IEEE International Professional Communication Conference Proceedings Piscataway, New Jersey.
- [9] Ericsson, K.A., Simon, H.A. (1980). "Verbal reports as data". *Psychological Review* 87: 215–251. doi:10.1037/0033-295X.87.3.215.
- [10] Ericsson, K.A., Simon, H.A. (1993). *Protocol analysis: Verbal reports as data (Revised edition)*. MIT Press Cambridge.

Penetration test

A **penetration test**, occasionally **pentest**, is a method of evaluating the security of a computer system or network by simulating an attack from a malicious source, known as a *Black Hat Hacker*, or *Cracker*. The process involves an active analysis of the system for any potential vulnerabilities that could result from poor or improper system configuration, both known and unknown hardware or software flaws, or operational weaknesses in process or technical countermeasures. This analysis is carried out from the position of a potential attacker and can involve active exploitation of security vulnerabilities. Any security issues that are found will be presented to the system owner, together with an assessment of their impact, and often with a proposal for mitigation or a technical solution. The intent of a penetration test is to determine the feasibility of an attack and the amount of business impact of a successful exploit, if discovered. It is a component of a full security audit. For example, the Payment Card Industry Data Security Standard (PCI DSS), and security and auditing standard, requires both annual and ongoing penetration testing (after system changes).

Black box vs. White box

Penetration tests can be conducted in several ways. The most common difference is the amount of knowledge of the implementation details of the system being tested that are available to the testers. Black box testing assumes no prior knowledge of the infrastructure to be tested. The testers must first determine the location and extent of the systems before commencing their analysis. At the other end of the spectrum, white box testing provides the testers with complete knowledge of the infrastructure to be tested, often including network diagrams, source code, and IP addressing information. There are also several variations in between, often known as grey box tests. Penetration tests can also be described as "full disclosure" (white box), "partial disclosure" (grey box), or "blind" (black box) tests based on the amount of information provided to the testing party.

The relative merits of these approaches are debated. Black box testing simulates an attack from someone who is unfamiliar with the system. White box testing simulates what might happen during an "inside job" or after a "leak" of sensitive information, where the attacker has access to source code, network layouts, and possibly even some passwords.

The services offered by penetration testing firms span a similar range, from a simple scan of an organization's IP address space for open ports and identification banners to a full audit of source code for an application.

Rationale

A penetration test should be carried out on any computer system that is to be deployed in a hostile environment, in particular any Internet facing site, before it is deployed. This provides a level of practical assurance that any malicious user will not be able to penetrate the system.

Black box penetration testing is useful in the cases where the tester assumes the role of an outside hacker and tries to intrude into the system without adequate knowledge of it.

Risks

Penetration testing can be an invaluable technique to any organization's information security program. Basic white box penetration testing is often done as a fully automated inexpensive process. However, black box penetration testing is a labor-intensive activity and requires expertise to minimize the risk to targeted systems. At a minimum, it may slow the organization's networks response time due to network scanning and vulnerability scanning. Furthermore, the possibility exists that systems may be damaged in the course of penetration testing and may be rendered inoperable, even though the organization benefits in knowing that the system could have been rendered

inoperable by an intruder. Although this risk is mitigated by the use of experienced penetration testers, it can never be fully eliminated.

Methodologies

The Open Source Security Testing Methodology Manual is a peer-reviewed methodology for performing security tests and metrics. The OSSTMM test cases are divided into five channels which collectively test: information and data controls, personnel security awareness levels, fraud and social engineering control levels, computer and telecommunications networks, wireless devices, mobile devices, physical security access controls, security processes, and physical locations such as buildings, perimeters, and military bases.

The OSSTMM focuses on the technical details of exactly which items need to be tested, what to do before, during, and after a security test, and how to measure the results. OSSTMM is also known for its Rules of Engagement which define for both the tester and the client how the test needs to properly run starting from denying false advertising from testers to how the client can expect to receive the report. New tests for international best practices, laws, regulations, and ethical concerns are regularly added and updated.

The National Institute of Standards and Technology (NIST) discusses penetration testing in SP800-115.^{[1] [2]} NIST's methodology is less comprehensive than the OSSTMM; however, it is more likely to be accepted by regulatory agencies. For this reason, NIST refers to the OSSTMM.

The Information Systems Security Assessment Framework (ISSAF) is a peer reviewed structured framework from the Open Information Systems Security Group that categorizes information system security assessment into various domains and details specific evaluation or testing criteria for each of these domains. It aims to provide field inputs on security assessment that reflect real life scenarios. The ISSAF should primarily be used to fulfill an organization's security assessment requirements and may additionally be used as a reference for meeting other information security needs. It includes the crucial facet of security processes and, their assessment and hardening to get a complete picture of the vulnerabilities that might exist. The ISSAF, however, is still in its infancy.

Standards and certification

The process of carrying out a penetration test can reveal sensitive information about an organization. It is for this reason that most security firms are at pains to show that they do not employ ex-black hat hackers and that all employees adhere to a strict ethical code. There are several professional and government certifications that indicate the firm's trustworthiness and conformance to industry best practice.

The Council of Registered Ethical Security Testers^[3] (CREST) is a UK non-profit association created to provide recognised standards and professionalism for the penetration testing industry.^[4] For organisations, CREST provides a provable validation of security testing methodologies and practices, aiding with client engagement and procurement processes and proving that the member company is able to provide testing services to the CREST standard. Three certifications are currently offered: the CREST Registered Tester and two CREST Certified Tester qualifications, one for infrastructure and one for application testing.^[5]

The Information Assurance Certification Review Board (IACRB) manages a penetration testing certification known as the Certified Penetration Tester (CPT). The CPT requires that the exam candidate pass a traditional multiple choice exam, as well as pass a practical exam that requires the candidate to perform a penetration test against live servers.

SANS provides a wide range of computer security training arena leading to a number of SANS qualifications. In 1999, SANS founded GIAC, the Global Information Assurance Certification, which according to SANS has been undertaken by over 20,000 members to date.^[6] Two of the GIAC certifications are penetration testing specific: the GIAC Certified Penetration Tester (GPEN) certification^[7] ; and the GIAC Web Application Penetration Tester (GWAPT) certification.^[8]

Offensive Security offers an Ethical Hacking certification (Offensive Security Certified Professional) - a training spin off of the BackTrack Penetration Testing distribution. The OSCP is a real-life penetration testing certification, requiring holders to successfully attack and penetrate various live machines in a safe lab environment. Upon completion of the course students become eligible to take a certification challenge, which has to be completed within twenty-four hours. Documentation must include procedures used and proof of successful penetration including special marker files.

Government-backed testing also exists in the US with standards such as the NSA Infrastructure Evaluation Methodology (IEM).

For web applications, the Open Web Application Security Project (OWASP) provides a framework of recommendations that can be used as a benchmark.

The Tiger Scheme offers two certifications: Qualified Tester (QST) and Senior Security Tester (SST). The SST is technically equivalent to CHECK Team Leader and QST is technically equivalent to the CHECK Team Member certification^[9]. Tiger Scheme certifies the individual, not the company.

The International Council of E-Commerce consultants certifies individuals in various e-business and information security skills. These include the Certified Ethical Hacker course, Computer Hacking Forensics Investigator program, Licensed Penetration Tester program and various other programs, which are widely available worldwide.

Web application penetration testing

Web application penetration testing refers to a set of services used to detect various security issues with web applications and identify vulnerabilities and risks, including:

- Known vulnerabilities in COTS applications
- Technical vulnerabilities: URL manipulation, SQL injection, cross-site scripting, back-end authentication, password in memory, session hijacking, buffer overflow, web server configuration, credential management, Clickjacking, etc,
- Business logic errors: Day-to-Day threat analysis, unauthorized logins, personal information modification, pricelist modification, unauthorized funds transfer, breach of customer trust etc.

OWASP, the Open Web Application Security Project, an open source web application security documentation project, has produced documents such as the OWASP Guide^[10] and the widely adopted OWASP Top 10^[11] awareness document.

The Firefox browser is a popular web application penetration testing tool, with many plugins^[12] specifically designed for web application penetration testing.

Damn vulnerable web app otherwise known as DVWA^[13] is an open source web application which has been made to be vulnerable so that security professionals and students can learn more about web application security.

Foundstone's Hacme Bank^[14] simulates a banking application. It helps developers and auditors practice web application attacks, including input validation flaws such as SQL injection and Cross Site Scripting (XSS).

References

- [1] Special Publication 800-42, Guideline on Network Security Testing (<http://csrc.nist.gov/publications/nistpubs/800-42/NIST-SP800-42.pdf>)
 - [2] Special Publication 800-115, Technical Guide to Information Security Testing and Assessment, September 2008 (replaces SP800-42) (<http://csrc.nist.gov/publications/nistpubs/800-115/SP800-115.pdf>)
 - [3] <http://www.crest-approved.org/index.html>
 - [4] "Infosec 2008: UK association of penetration testers launched" (<http://www.computerweekly.com/Articles/2008/04/24/230417/infosec-2008-uk-association-of-penetration-testers.htm>). Computer Weekly. 2008-04-24. . Retrieved 2008-08-16.
 - [5] King, Leo (2008-04-24). "Security testing standards council launched" (<http://www.computerworlduk.com/management/security/cybercrime/news/index.cfm?newsid=8730>). Computerworld UK. . Retrieved 2008-08-16.
 - [6] http://www.sans.org/why_certify.php SANS Institute
 - [7] <http://giac.org/certifications/security/GPEN.php>
 - [8] <http://giac.org/certifications/security/GWAPT.php>
 - [9] http://www.cesg.gov.uk/products_services/iacs/check/index.shtml
 - [10] http://www.owasp.org/index.php/OWASP_Guide_Project
 - [11] http://www.owasp.org/index.php/OWASP_Top_Ten_Project
 - [12] <https://addons.mozilla.org/en-US/firefox/collection/webappsec>
 - [13] <http://www.dvwa.co.uk/>
 - [14] <http://www.foundstone.com/us/resources/proddesc/hacmebank.htm>
- Johnny Long; *Google Hacking for Penetration Testers*, Syngress, 2007. ISBN 978-1597491761
 - Stuart McClure; *Hacking Exposed: Network Security Secrets and Solutions*, McGraw-Hill, 2009. ISBN 978-0071613743

External links

- List of Network Penetration Testing software (<https://mosaicsecurity.com/categories/27-network-penetration-testing>), Mosaic Security Research

Performance testing

Performance Testing covers a broad range of engineering or functional evaluations where a material, product, system, or person is not specified by detailed material or component specifications: rather, emphasis is on the final measurable performance characteristics. Testing can be a qualitative or quantitative procedure.

Performance testing can refer to the assessment of the performance of a human examinee. For example, a behind-the-wheel driving test is a performance test of whether a person is able to perform the functions of a competent driver of an automobile.

In the computer industry, software performance testing is used to determine the speed or effectiveness of a computer, network, software program or device. This process can involve quantitative tests done in a lab, such as measuring the response time or the number of MIPS (millions of instructions per second) at which a system functions. Qualitative attributes such as reliability, scalability and interoperability may also be evaluated. Performance testing is often done in conjunction with stress testing.

Examples

- Building and Construction Performance Testing
 - Fire protection (ASTM D176)
 - Packaging Performance (hazardous materials, dangerous goods, ASTM D4169)
 - Performance Index for Tires (ASTM F538)
 - Personal protective equipment performance
 - Performance test (bar exam) for lawyers
 - Proficiency Testing, Performance test (assessment)
 - Several Defense Standards
 - Software performance testing, Web testing
 - Wear of Textiles (ASTM D123)
 - and many others.
-

PlanetLab

PlanetLab is a group of computers available as a testbed for computer networking and distributed systems research. It was established in 2002 by Prof. Larry L. Peterson, and as of June 2010 was composed of 1090 nodes at 507 sites worldwide. Each research project has a "slice", or virtual machine access to a subset of the nodes.

Accounts are limited to persons affiliated with corporations and universities that host PlanetLab nodes. However, a number of free, public services have been deployed on PlanetLab, including CoDeeN, the Coral Content Distribution Network, and Open DHT ^[1]. Open DHT was taken down on 1 July 2009.

PlanetLab members develop tools for the greater good of the community, and as a result each user has a wide choice of tools to use in order to complete regular slice maintenance tasks.

PlanetLab experiences have been critical in the formulation of the US National Science Foundation's Global Environment for Network Innovations (GENI) initiative.

External links

- PlanetLab ^[2]
- PlanetLab Europe (PLE) ^[3]
- PlanetLab-Wiki ^[4]

References

- [1] <http://opendht.org/>
- [2] <http://www.planet-lab.org/>
- [3] <http://www.planet-lab.eu/>
- [4] <https://wiki.planet-lab.org/twiki/bin/view/Planetlab>
-

Playtest

A **playtest** is the process by which a game designer tests a new game for bugs and flaws before bringing it to market. Playtests can be run "open", "closed", "beta", or otherwise.

Playtests are very common with computer games, board games and role-playing games, where they have become an established part of the quality control process.

An open playtest could be considered open to anyone who wishes to join, or it may refer to a game company's recruiting testers from outside. Closed is an internal testing process not available to the public. Beta testing normally refers to the final stages of testing just prior to going to market with a product and is usually run semi-open with a limited form of the game in order to find any last-minute problems.

The playtest concept has even carried over into a full-fledged sport. Jim Foster, inventor and founder of the Arena Football League, tested his concept of indoor football in a special one-time game in 1986. The Rockford Metros and the Chicago Politicians, played the game in Rockford, Illinois. The test proved successful, and four teams began the league's first season the following year.

Portability testing

Portability testing refers to the process of testing the ease with which a computer software component can be moved from one environment to another, e.g. moving from Windows 2000 to Windows XP. This is typically measured in terms of the maximum amount of effort permitted. Results are expressed in terms of the time required to move the software and complete data conversion and documentation updates.

Probe effect

Probe effect is unintended alteration in system behavior caused by measuring that system. In code profiling and performance measurements, the delays introduced by insertion/removal of code instrumentation may result in a non-functioning application, or unpredictable behavior.

Examples

In electronics, by attaching a multimeter, oscilloscope, or other probing device, small amounts of capacitance, resistance, or inductance may be introduced. Though good scopes have very slight effects, in sensitive circuitry these can lead to unexpected failures, or conversely, unexpected fixes to failures.

In debugging of parallel computer programs, sometimes failures (such as deadlocks) are not present when debugger's code (which was meant to help finding a reason for deadlocks by visualising points of interest in the program code) is attached to the program. This is because additional code changed timing of the execution of parallel processes, and because of that deadlocks were avoided.^[1]

Sources

[1] Event manipulation for Nondeterministic Shared-Memory Programs (http://books.google.com/books?id=vOE0s6Zfk6gC&pg=PA287&dq=Probe+effect&hl=en&ei=I8pyTLHVDMuTjAeikvD6CA&sa=X&oi=book_result&ct=result&resnum=4&ved=0CDcQ6AEwAw#v=onepage&q=Probe+effect&f=false) / High-Performance Computing and Networking. 9th International Conference, HPCN Europe 2001, Amsterdam, The Netherlands, June 25–27, 2001,

Program mutation

For the biological term, see: Gene mutation analysis.

Mutation testing (or *Mutation analysis* or *Program mutation*) is a method of software testing, which involves modifying programs' source code or byte code in small ways.^[1] A test suite that does not detect and reject the mutated code is considered defective. These so-called *mutations*, are based on well-defined *mutation operators* that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as driving each expression to zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.

Aim

Tests can be created to verify the correctness of the implementation of a given software system, but the creation of tests still poses the question whether the tests are correct and sufficiently cover the requirements that have originated the implementation. (This technological problem is itself an instance of a deeper philosophical problem named "Quis custodiet ipsos custodes?" ["Who will guard the guards?"].) In this context, mutation testing was pioneered in the 1970s to locate and expose weaknesses in test suites. The theory was that if a mutation was introduced without the behavior (generally output) of the program being affected, this indicated either that the code that had been mutated was never executed (redundant code) or that the testing suite was unable to locate the injected fault. In order for this to function at any scale, a large number of mutations had to be introduced into a large program, leading to the compilation and execution of an extremely large number of copies of the program. This problem of the expense of mutation testing had reduced its practical use as a method of software testing, but the increased use of object oriented programming languages and unit testing frameworks has led to the creation of mutation testing tools for many programming languages as a means to test individual portions of an application.

Historical overview

Mutation testing was originally proposed by Richard Lipton as a student in 1971,^[2] and first developed and published by DeMillo, Lipton and Sayward. The first implementation of a mutation testing tool was by Timothy Budd as part of his PhD work (titled *Mutation Analysis*) in 1980 from Yale University.

Recently, with the availability of massive computing power, there has been a resurgence of mutation analysis within the computer science community, and work has been done to define methods of applying mutation testing to object oriented programming languages and non-procedural languages such as XML, SMV, and finite state machines.

In 2004 a company called Certess Inc. extended many of the principles into the hardware verification domain. Whereas mutation analysis only expects to detect a difference in the output produced, Certess extends this by verifying that a checker in the testbench will actually detect the difference. This extension means that all three stages of verification, namely: activation, propagation and detection are evaluated. They have called this functional qualification.

Fuzzing is a special area of mutation testing. In fuzzing, the messages or data exchanged inside communication interfaces (both inside and between software instances) are mutated, in order to catch failures or differences in processing the data. Codenomicon^[3] (2001) and Mu Dynamics (2005) evolved fuzzing concepts to a fully stateful mutation testing platform, complete with monitors for thoroughly exercising protocol implementations.

Mutation testing overview

Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a *mutant*. If the test suite is able to detect the change (i.e. one of the tests fails), then the mutant is said to be *killed*.

For example, consider the following C++ code fragment:

```
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

The condition mutation operator would replace `&&` with `||` and produce the following mutant:

```
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

Now, for the test to kill this mutant, the following condition should be met:

- Test input data should cause different program states for the mutant and the original program. For example, a test with `a = 1` and `b = 0` would do this.
- The value of 'c' should be propagated to the program's output and checked by the test.

Weak mutation testing (or *weak mutation coverage*) requires that only the first condition is satisfied. *Strong mutation testing* requires that both conditions are satisfied. Strong mutation is more powerful, since it ensures that the test suite can really catch the problems. Weak mutation is closely related to code coverage methods. It requires much less computing power to ensure that the test suite satisfies weak mutation testing than strong mutation testing.

Equivalent mutants

Many mutation operators can produce equivalent mutants. For example, consider the following code fragment:

```
int index = 0;

while (...)
{
    ...;
    index++;

    if (index == 10) {
        break;
    }
}
```

Boolean relation mutation operator will replace `==` with `>=` and produce the following mutant:

```
int index = 0;
```



```
while (...)
{
    ...;
    index++;

    if (index >= 10) {
        break;
    }
}
```

However, it is not possible to find a test case that could kill this mutant. The resulting program is equivalent to the original one. Such mutants are called *equivalent mutants*.

Equivalent mutants detection is one of biggest obstacles for practical usage of mutation testing. The effort needed to check if mutants are equivalent or not, can be very high even for small programs.^[4]

Mutation operators

A variety of mutation operators were explored by researchers. Here are some examples of mutation operators for imperative languages:

- Statement deletion.
- Replace each boolean subexpression with *true* and *false*.
- Replace each arithmetic operation with another one, e.g. + with *, - and /.
- Replace each boolean relation with another one, e.g. > with >=, == and <=.
- Replace each variable with another variable declared in the same scope (variable types should be the same).

These mutation operators are also called traditional mutation operators. Beside this, there are mutation operators for object-oriented languages^[5], for concurrent constructions^[6], complex objects like containers^[7] etc. They are called class-level mutation operators. For example the MuJava tool offers various class-level mutation operators such as: Access Modifier Change, Type Cast Operator Insertion, Type Cast Operator Deletion. Moreover, mutation operators have been developed to perform security vulnerability testing of programs^[8]

References

- [1] A Practical System for Mutation Testing: Help for the Common Programmer (<http://cs.gmu.edu/~offutt/rsrch/papers/practical.pdf>) by A. Jefferson Offutt.
- [2] Mutation 2000: Uniting the Orthogonal (<http://cs.gmu.edu/~offutt/rsrch/papers/mut00.pdf>) by A. Jefferson Offutt and Roland H. Untch.
- [3] Kaksonen, Rauli. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. 2001. (<http://www.codenomicon.com/resources/publications.shtml>)
- [4] P. G. Frankl, S. N. Weiss, and C. Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38:235–253, 1997.
- [5] MuJava: An Automated Class Mutation System (<http://www.cs.gmu.edu/~offutt/rsrch/papers/mujava.pdf>) by Yu-Seung Ma, Jeff Offutt and Yong Rae Kwo.
- [6] Mutation Operators for Concurrent Java (J2SE 5.0) (http://www.irisa.fr/manifestations/2006/Mutation2006/papers/14_Final_version.pdf) by Jeremy S. Bradbury, James R. Cordy, Juergen Dingel.
- [7] Mutation of Java Objects (<http://www.cs.colostate.edu/~bieman/Pubs/AlexanderBiemanGhoshJiISSRE02.pdf>) by Roger T. Alexander, James M. Bieman, Sudipto Ghosh, Bixia Ji.
- [8] Mutation-based Testing of Buffer Overflows, SQL Injections, and Format String Bugs (<http://qspace.library.queensu.ca/handle/1974/1359>) by H. Shahriar and M. Zulkernine.

Further reading

- Aristides Dasso, Ana Funes (2007). *Verification, Validation and Testing in Software Engineering*. Idea Group Inc. ISBN 1591408512. See Ch. VII *Test-Case Mutation* for overview on mutation testing.
- Paul Ammann, Jeff Offutt (2008). *Introduction to Software Testing*. Cambridge University Press. ISBN 0-52188-038-1. See Ch. V *Syntax Testing* for an overview of mutation testing.
- Yue Jia, Mark Harman (September 2009). "An Analysis and Survey of the Development of Mutation Testing" (<http://www.dcs.kcl.ac.uk/pg/jiayue/repository/TR-09-06.pdf>) (PDF). *CREST Centre, King's College London, Technical Report TR-09-06*.

External links

- Mutation testing (<http://cs.gmu.edu/~offutt/rsrch/mut.html>) list of tools and publications by Jeff Offutt.
- Mutation Testing Repository (<http://www.dcs.kcl.ac.uk/pg/jiayue/repository/>) A publication repository that aims to provide a full coverage of the publications in the literature on Mutation Testing.
- Jumble (<http://jumble.sourceforge.net/>) Bytecode based mutation testing tool for Java
- PIT (<http://pitest.org/>) Bytecode based mutation testing tool for Java
- Jester (<http://jester.sourceforge.net/>) Source based mutation testing tool for Java
- Heckle (<http://glu.ttono.us/articles/2006/12/19/tormenting-your-tests-with-heckle>) Mutation testing tool for Ruby
- Nester (<http://nester.sourceforge.net/>) Mutation testing tool for C#
- Mutagenesis (<https://github.com/padraic/mutagenesis>) Mutation testing tool for PHP

Protocol implementation conformance statement

A **Protocol Implementation Conformance Statement** or most commonly *PICS* is a structured document which asserts which specific requirements are met by a given implementation of a protocol standard. It is often completed as a record of formal protocol conformance test results, and some automated test systems machine-author a PICS as output. A potential buyer or user of the implementation can consult the PICS to determine if it meets his or her requirements.

ETSI publish an example proforma PICS ^[1] showing which information would be completed.

References

[1] http://portal.etsi.org/mbs/Referenced%20Documents/ts_101_823_02.pdf

Pseudolocalization

Pseudolocalization is a software testing method that is used to test internationalization aspects of software. Specifically, it brings to light potential difficulties with localization by replacing localizable text (particularly in a graphical user interface) with text that imitates the most problematic characteristics of text from a wide variety of languages, and by forcing the application to deal with similar input text.

If used properly, it provides a cheap but effective sanity test for localizability that can be helpful in the early stages of a software project.

Rationale

If software is not designed with localizability in mind, certain problems can occur when the software is localized. Text in a target language may tend to be significantly longer than the corresponding text in the original language of the program, causing the ends of text to be cut off if insufficient space is allocated. Words in a target language may be longer, causing awkward line breaks. In addition, individual characters in a target language may require more space, causing modified characters to be cut off vertically, for example. Even worse, characters of a target language may fail to render properly (or at all) if support for an appropriate font is not included. (This is a larger problem for legacy software than for newer programs.) On the input side, programmers may make inappropriate assumptions about the form that user input can take.

Method

For small changes to mature software products, for which a large amount of target text is already available, directly testing several target languages may be the best option. For newer software (or for larger user-interface changes), however, waiting for text to be translated can introduce a significant lag into the testing schedule. In addition, it may not be cost-effective to translate UI text early in the development cycle, as it might change and need to be retranslated. Here, pseudolocalization can be the best option, as no real translation is needed.

Typically, pseudolocalized text (pseudo-translation) for a program will be generated and used as if it were for a real locale. Pseudolocalized text should be longer than the original text (perhaps twice as long), contain longer unbroken strings of characters to test line breaking, and contain characters from different writing systems. A tester will then inspect each element of the UI to make sure everything is displayed properly. To make it easier for the tester to find his or her way around the UI, the text may include the original text, or perhaps characters that look similar to the original text. For example, the string:

```
Edit program settings
```

might be replaced with:

```
[!!! εĐiț Pr0đRãm sӨTт1Иђ$ !!!]
```

The brackets on either side of the text helps to spot the following issues:

- text that is cut off
- concatenated strings
- hard-coded strings

This type of transformation can be performed by a simple tool and does not require a human translator, resulting in time and cost savings.

Alternatively, a machine translation system can be used for automatically generating translated strings. This type of machine-generated pseudolocalization has the advantage of the translated strings featuring the characteristics specific to the target language and being available in real time at very low cost.

One approach to automatically generating translated strings is to add non-ASCII characters at the beginning and end of the existing text. This allows the existing text to still be read, but clearly identifies what text has been externalized and what text has not been externalized and exposes UI issues such as the need to accommodate longer text strings. This allows regular QA staff to test that the code has been properly internationalized.

Tools such as Alchemy Catalyst from Alchemy Software Development and SDL Passolo from SDL have advanced pseudo translation/localization capability including ability to view rendered Pseudolocalized dialog's and forms in the tools themselves for formats such as .net, wpf .rc .dll and .exe.

References

Engineering Windows 7 for a Global Market ^[1]

[1] <http://blogs.msdn.com/b/e7/archive/2009/07/07/engineering-windows-7-for-a-global-market.aspx>

Pychecker

PyChecker is a source code bug checker for the Python programming language.^[1]

References

[1] PyChecker: Python Code Analysis (<http://www.blog.pythonlibrary.org/2011/01/26/pychecker-python-code-analysis/>)

External links

- Home site (<http://pychecker.sourceforge.net/>)
- PyLint, PyChecker or PyFlakes ? (<http://stackoverflow.com/questions/1428872/pylint-pychecker-or-pyflakes>)

Pylint

Pylint is a source code bug and quality checker for the Python programming language. It follows the style recommended by PEP 8, the Python style guide.^[1]

References

[1] Style Guide for Python Code (<http://www.python.org/dev/peps/pep-0008/>)

External links

- Home site (<http://pypi.python.org/pypi/pylint>)
- Pylint: The Python Code bug/quality checker (<http://the-moni-blog.blogspot.com/2010/01/pylint-python-code-bugquality-checker.html>)
- pychecker vs pylint vs Django (<http://lukeplant.me.uk/blog/posts/pychecker-vs-pylint-vs-django/>)

User:Mbarberony/sandbox

In the context of software engineering, software quality refers to **two related but distinct notions** that exist wherever quality is defined in a business context:

- **Software functional quality** reflects how well it complies/conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the **fitness for purpose** of a piece of software or how it compares to competitors in the marketplace as a worthwhile product^[1];
- **Software structural quality** refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

Structural quality is evaluated through the analysis of the software inner structure, its source code, in effect how its architecture adheres to sound principles of software architecture. In contrast, **functional quality** is typically enforced and measured through software testing.

Historically, the structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the **ISO 9126-3** and the subsequent **25000:2005**^[2] quality model. Based on these models, the software structural quality characteristics have been clearly defined by the **Consortium for IT Software Quality (CISQ)**, an independent organization founded by the **Software Engineering Institute (SEI)**^[3] at Carnegie Mellon University, and the **Object Management Group (OMG)**^[4].

CISQ has defined 5 major desirable characteristics needed for a piece of software to provide business value: **Reliability, Efficiency, Security, Maintainability** and (adequate) **Size**.

Software quality measurement is about quantifying to what extent a software or system rates along each of these five dimensions. An aggregated measure of software quality can be computed through a qualitative or a quantitative scoring scheme or a mix of both and then a weighting system reflecting the priorities. This view of software quality being positioned on a linear continuum has to be supplemented by the analysis of **critical vulnerabilities** that under specific circumstances can lead to catastrophic outages or performance degradations that make a given system unsuitable for use regardless of rating based on aggregated measurements.

Motivation for Defining Software Quality

'A science is as mature as its measurement tools,' (Louis Pasteur in Ebert and al.1, p. 91 and software engineering has evolved to a level of maturity that makes it not only possible but also necessary to measure quality software for at least two reasons:

- **Risk Management:** Software failure has caused more than inconvenience. Software errors have caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper ^[5]. This resulted in requirements for the development of some types of software, particularly and historically for software embedded in medical and other devices that regulate critical infrastructures: "(When engineers that write embedded software) see Java programs stalling for one third of a second to perform garbage collection and update the user interface, and they envision airplanes falling out of the sky." ^[6]). In the United States, within the [Federal Aviation Administration|Federal Aviation Administration (FAA)], the Aircraft Certification Service ^[7] provides software programs, policy, guidance and training, focus on software and Complex Electronic Hardware that has an effect on the airborne product (a "product" is an aircraft, an engine, or a propeller)".
- **Cost Management:** As in any other fields of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs. Industry data demonstrate that poor application structural quality in core business applications (such as Enterprise Resource Planning (ERP) ^[8], Relationship Management (CRM) ^[9] or large transaction processing systems in financial services) results in cost and schedule overruns and creates waste in the form of rework (up to 45% of development time in some organizations ^[10]). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems.

However, the distinction between motivation for measuring and improving **software quality in embedded system** (with emphasis on risk management) and **software quality in business software** (with emphasis on cost and maintainability management) is becoming somewhat irrelevant. Embedded systems now often include user interface and their designers are as much concerned with issues affecting usability and user productivity as their counterparts who focus on business applications. The latter are in turn looking at ERP or CRM system as a corporate nervous system whose uptime and performance are vital to the well-being of the enterprise. This convergence is most visible in mobile computing: a user who accesses an ERP application on her smartphone is depending on the quality of software across all types of software layers.

Both types of software now use multi-layered technology stacks and complex architecture so software quality analysis and measurement have to be managed in a comprehensive and consistent manner, decoupled from the software ultimate purpose or use. In both cases, engineers and management need to be able to make rational decisions based on measurement and fact-based analysis in adherence to the precept that "*In God (we) trust. All others bring data*". ((mis-)attributed to W. Edwards Deming and others).

Definition

Even though (as noted in the article on quality in business) "*quality is a perceptual, conditional and somewhat subjective attribute and may be understood differently by different people,*" **Software structural quality** characteristics have been clearly defined by the **Consortium for IT Software Quality (CISQ)**, an independent organization founded by the Software Engineering Institute (SEI) at Carnegie Mellon University (<http://www.sei.cmu.edu>), and the Object Management Group (OMG) (<http://www.omg.org>). Under the guidance of Bill Curtis, co-author of the Capability Maturity Model framework and CISQ's first Director and Capers Jones, CISQ's Distinguished Advisor, CISQ has defined 5 major desirable characteristics of a piece of software needed to provide business value. In the House of Quality model, these are "Whats" that need to be achieved:

- **Reliability:** An attribute of resiliency and structural solidity. Reliability measures the level of risk and the likelihood of potential application failures. It also measures the defects injected due to modifications made to the software (its “stability” as termed by ISO). The goal for checking and monitoring Reliability is to reduce and prevent application downtime, application outages and errors that directly affect users, and enhance the image of IT and its impact on a company’s business performance.
- **Efficiency:** The source code and software architecture attributes are the elements that ensure high performance once the application is in run-time mode. Efficiency is especially important for applications in high execution speed environments such as algorithmic or transactional processing where performance and scalability are paramount. An analysis of source code efficiency and scalability provides a clear picture of the latent business risks and the harm they can cause to customer satisfaction due to response-time degradation.
- **Security:** A measure of the likelihood of potential security breaches due to poor coding and architectural practices. This quantifies the risk of encountering critical vulnerabilities that damage the business.
- **Maintainability:** Maintainability includes the notion of adaptability, portability and transferability (from one development team to another). Measuring and monitoring maintainability is a must for mission-critical applications where change is driven by tight time-to-market schedules and where it is important for IT to remain responsive to business-driven changes. It is also essential to keep maintenance costs under control.
- **Size:** While not a quality attribute per se, the sizing of source code is a software characteristic that obviously impacts maintainability. Combined with the above quality characteristics, software size can be used to assess the amount of work produced and to be done by teams, as well as their productivity through correlation with time-sheet data, and other SDLC-related metrics.

Software functional quality is defined as conformance to explicitly stated functional requirements, identified for example using **Voice of the Customer** analysis (part of the **Design for Six Sigma** toolkit and/or documented through **use cases**) and the level of satisfaction experienced by end-users. The later is referred as to as **usability** and is concerned with how intuitive and responsive the **user interface** is, how easy simple and complex operations can be performed, how useful **error messages** are. Typically, software testing practices and tools insure that a piece of software behaves in compliance with the original design, planned user experience and desired testability, ie a software's disposition to support acceptance criteria.

Also, the availability of (free or paid) support may factor into the usability of the software.

The dual structural/functional dimension of software quality is consistent with the model proposed in **Steve McConnell's Code Complete** which divides software characteristics into two pieces: internal and external quality characteristics. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not ^[11].

Alternative Approaches to Software Quality Definition

One of the challenges in defining quality is that *"everyone feels they understand it"* ^[12] and other definitions of software quality could be based on extending the various description of the concept of quality used in business (see a list of possible definition here.)

Dr. **Tom DeMarco** has proposed that *"a product's quality is a function of how much it changes the world for the better."* ^[13]. This can be interpreted as meaning that functional quality and user satisfaction, is more important than structural quality in determining software quality.

Another definition, coined by **Gerald Weinberg** in *Quality Software Management: Systems Thinking*, is *"Quality is value to some person."* This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as *"Who are the people we want to value our software?"* and *"What will be valuable to them?"*

Software Quality Measurement

Although the concepts presented in this section are applicable to both Software Structural and Functional Quality, measurement of the latter is essentially performed through testing, see main article: Software Testing.

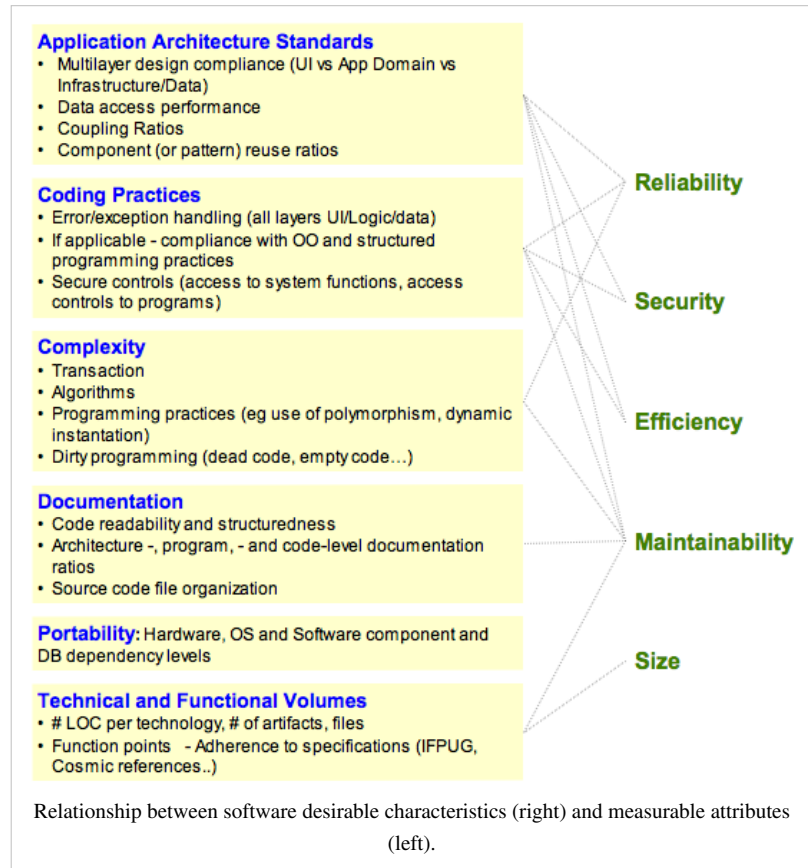
Introduction

Software quality measurement is about quantifying to what extent a software or system possesses desirable characteristics. This can be performed through qualitative or quantitative means or a mix of both. In both cases, for each **desirable characteristic**, there are a set of **measurable attributes** the existence of which in a piece of software or system tend to be correlated and associated to this characteristic. For example, an attribute associated with portability is the number of target-dependent statements in a program. More precisely, using the **Quality Function Deployment** approach, these measurable attributes are the "Hows" that need to be enforced to enable the "whats" in the Software Quality definition above.

The structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the **ISO 9126-3** and the subsequent **25000:2005** ^[2] quality model. The main focus is on internal structural quality. Subcategories have been created to handle specific areas like business application architecture and technical characteristics such as data access and manipulation or the notion of transactions.

The dependence tree between software quality characteristics and their measurable attributes is represented in the diagram on the right, where each of the 5 characteristics that matter for the user (right) or owner of the business system depends on measurable attributes (left):

- Application Architecture Practices
- Coding Practices
- Application Complexity
- Documentation
- Portability
- Technical & Functional Volume



Code-Based Analysis of Software Quality Attributes

Many of the existing software measures count structural elements of the application that result from parsing the source code such individual instructions (Park, 1992)^[14], tokens (Halstead, 1977)^[15], control structures (McCabe, 1976), and objects (Chidamber & Kemerer, 1994)^[16].

Software quality measurement is about quantifying to what extent a software or system rate along these dimensions. The analysis can be performed using a qualitative, quantitative approach or a mix of both to provide an aggregate view (using for example weighted average(s) that reflect relative importance between the factor being measured).

This view of software quality on a linear continuum has to be supplemented by the identification of **discrete critical vulnerabilities**. These vulnerabilities may not fail a test case, but they are the result of bad practices that under specific circumstances can lead to catastrophic outages, performance degradations, security breaches, corrupted data, and myriad other problems (Nygard, 2007)^[17] that makes a given system de facto unsuitable for use regardless of its rating based on aggregated measurements. A well known example of vulnerability is the **Common Weakness Enumeration** at <http://cwe.mitre.org/>(Martin, 2001)^[18], a repository of vulnerabilities in the source code that make applications exposed to security breaches.

The measurement of critical application characteristics involves measuring structural attributes of the application's architecture, coding, in-line documentation, as displayed in the picture above. Thus, each characteristic is affected by attributes at numerous levels of abstraction in the application and all of which must be included calculating the characteristic's measure if it is to be a valuable predictor of quality outcomes that affect the business. The **layered approach** to calculating characteristic measures displayed in the figure above was first proposed by Boehm and his colleagues at TRW (Boehm, 1978)^[19] and is the approach taken in the ISO 9126 and 25000 series standards. These attributes can be measured from the parsed results of a static analysis of the application source code. Even dynamic characteristics of applications such as reliability and performance efficiency have their causal roots in the static structure of the application.

Note that the structural quality analysis and measurement is performed throughs the analysis of the source code, the architecture, software framework, database schema in relationship to principles and standards that together define the conceptual and logical architecture of a system. This is distinct from the basic, local, component-level code analysis typically performed by development tools which are mostly concerned with implementation considerations and are crucial during debugging and testing activities.

Measuring Reliability

The root causes of poor reliability are found in a combination of non- compliance with good architectural and coding practices. This non-compliance can be detected by measuring the static quality attributes of an application. Assessing the static attributes underlying an application's reliability provides an estimate of the level of business risk and the likelihood of potential application failures and defects the application will experience when placed in operation.

Assessing reliability requires checks of at least the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Coding Practices
- Complexity of algorithms
- Complexity of programming practices
- Compliance with Object-Oriented and Structured Programming best practices (when applicable)
- Component or pattern re-use ratio
- Dirty programming
- Error & Exception handling (for all layers - GUI, Logic & Data)
- Multi-layer design compliance
- Resource bounds management
- Software avoids patterns that will lead to unexpected behaviors
- Software manages data integrity and consistency
- Transaction complexity level

Depending on the application architecture and the third-party components used (such as external libraries or frameworks), custom checks should be defined along the lines drawn by the above list of best practices to ensure a better assessment of the reliability of the delivered software.

Measuring Efficiency

As with Reliability, the causes of performance inefficiency are often found in violations of good architectural and coding practice which can be detected by measuring the static quality attributes of an application. These static attributes predict potential operational performance bottlenecks and future scalability problems, especially for applications requiring high execution speed for handling complex algorithms or huge volumes of data.

Assessing performance efficiency requires checking at least the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Appropriate interactions with expensive and/or remote resources
- Data access performance and data management
- Memory, network and disk space management
- Coding Practices
- Compliance with Object-Oriented and Structured Programming best practices (as appropriate)
- Compliance with SQL programming best practices

Measuring Security

Most security vulnerabilities result from poor coding and architectural practices such as SQL injection or cross-site scripting. These are well documented in lists maintained by CWE <http://cwe.mitre.org/> (see below), and the SEI/Computer Emergency Center (CERT) at Carnegie Mellon University.

Assessing security requires at least checking the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Multi-layer design compliance
- Security best practices (Input Validation, SQL Injection, Cross-Site Scripting, etc. See CWE's Top 25 <http://www.sans.org/top25-programming-errors/>)
- Programming Practices (code level)
- Error & Exception handling
- Security best practices (system functions access, access control to programs)

Measuring Maintainability

Maintainability includes concepts of modularity, understandability, changeability, testability, reusability, and transferability from one development team to another. These do not take the form of critical issues at the code level. Rather, poor maintainability is typically the result of thousands of minor violations with best practices in documentation, complexity avoidance strategy, and basic programming practices that make the difference between clean and easy-to-read code vs. unorganized and difficult-to-read code.

Assessing maintainability requires checking the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Architecture, Programs and Code documentation embedded in source code
- Code readability
- Complexity level of transactions
- Complexity of algorithms
- Complexity of programming practices
- Compliance with Object-Oriented and Structured Programming best practices (when applicable)
- Component or pattern re-use ratio
- Controlled level of dynamic coding
- Coupling ratio
- Dirty programming
- Documentation
- Hardware, OS, middleware, software components and database independence
- Multi-layer design compliance
- Portability
- Programming Practices (code level)
- Reduced duplicated code and functions
- Source code file organization cleanliness

Measuring Size

Measuring software size requires that the whole source code be correctly gathered, including database structure scripts, data manipulation source code, component headers, configuration files etc. There are essentially two types of software sizes to be measured, the **technical size (footprint)** and the **functional size**:

- There are several software technical sizing methods that have been widely described here: http://en.wikipedia.org/wiki/Software_Sizing. The most common technical sizing method is number of Lines Of Code (#LOC) per technology, number of files, functions, classes, tables, etc, from which backfiring Function Points can be computed;
- The most common for measuring functional size is **Function Point Analysis** http://en.wikipedia.org/wiki/Function_point#See_also. Function Point Analysis measures the size of the software deliverable from a user's perspective. Function Point sizing is done based on user requirements and provides an accurate representation of both size for the developer/estimator and value (functionality to be delivered) and reflects the business functionality being delivered to the customer. The method includes the identification and weighting of user recognizable inputs, outputs and data stores. The size value is then available for use in conjunction with numerous measures to quantify and to evaluate software delivery and performance (Development Cost per Function Point; Delivered Defects per Function Point; Function Points per Staff Month..).

The Function Point Analysis sizing standard is supported by the **International Function Point Users Group** (FFPUG) (www.ifpug.org). It can be applied early in the software development lifecycle and it is not dependent on lines of code like the somewhat inaccurate Backfiring method. The method is technology agnostic and can be used for comparative analysis across organizations and across industries.

Since the inception of Function Point Analysis, several variations have evolved and the family of functional sizing techniques has broadened to include such sizing measures as COSMIC , NESMA, Use Case Points, FP Lite, Early and Quick FPs, and most recently Story Points. However, Function Points has a history of statistical accuracy, and has been used as a common unit of work measurement in numerous application development management (ADM) or outsourcing engagements, serving as the 'currency' by which services are delivered and performance is measured.

One common limitation to the Function Point methodology is that it is a manual process and therefore it can be labor intensive and costly in large scale initiatives such as application development or outsourcing engagements. This negative aspect of applying the methodology may be what motivated industry IT leaders to form the Consortium for IT Software Quality (www.it-cisq.org) focused on introducing a computable metrics standard for automating the measuring of software size while the IFPUG www.ifpug.org keep promoting a manual approach as most of its activity rely on FP counters certifications.

Identifying Critical Vulnerabilities

Critical Vulnerabilities are specific architectural and/or coding bad practices that result in the highest, immediate or long term, business disruption risk.

These are quite often technology-related and depend heavily on the context, business objectives and risks. Some may consider respect for naming conventions while others – those preparing the ground for a knowledge transfer for example – will consider it as absolutely critical.

Critical Violations can also be classified per CISQ Characteristics. Basic example below:

- Reliability
 - Avoid software patterns that will lead to unexpected behavior (Uninitialized variable, null pointers, etc.)
 - Methods, procedures and functions doing Insert, Update, Delete, Create Table or Select must include error management
 - Multi-thread functions should be made thread safe, for instance servlets or strutsstruts action classes must not have instance/non-final static fields
- Efficiency
 - Ensure centralization of client requests (incoming and data) to reduce network traffic
 - Avoid SQL queries that don't use an index against large tables in a loop
- Security
 - Avoid fields in servlet classes that are not final static
 - Avoid data access without including error management
 - Check control return codes and implement error handling mechanisms
 - Ensure input validation to avoid cross-site scripting flaws or SQL injections flaws
- Maintainability
 - Deep inheritance trees and nesting should be avoided to improve comprehensibility
 - Modules should be loosely coupled (fanout, intermediaries,) to avoid propagation of modifications
 - Enforce homogeneous naming conventions

References

Notes

- [1] Pressman, p. 388
- [2] http://webstore.iec.ch/preview/info_isoiec25000%7Bed1.0%7Den.pdfISO
- [3] <http://www.sei.cmu.edu>
- [4] <http://www.omg.org>
- [5] <http://sunnyday.mit.edu/papers/therac.pdf> (PDF)
- [6] <http://ptolemy.eecs.berkeley.edu/publications/papers/02/embsoft/embsoftwre.pdf>
- [7] http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/FAA
- [8] http://en.wikipedia.org/wiki/Enterprise_resource_planning
- [9] http://en.wikipedia.org/wiki/Customer_Relationship_ManagementCustomer
- [10] <http://www.ralphyoung.net/conferences/ImprovingQuality2.ppt>
- [11] DeMarco, T., *Management Can Make Quality (Im)possible*, Cutter IT Summit, Boston, April 1999
- [12] Crosby, P., *Quality is Free*, McGraw-Hill, 1979

- [13] http://en.wikipedia.org/w/index.php?title=Software_quality&diff=436729813&oldid=434635568#CITEREFMcConnell1993
- [14] Park, R.E. (1992). *Software Size Measurement: A Framework for Counting Source Statements*. (CMU/SEI-92-TR-020). Software Engineering Institute, Carnegie Mellon University
- [15] Halstead, M.E. (1977). *Elements of Software Science*. Elsevier North-Holland.
- [16] Chidamber, S. & C. Kemerer. C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20 (6), 476-493
- [17] Nygard, M.T. (2007). *Release It! Design and Deploy Production Ready Software*. The Pragmatic Programmers.
- [18] Martin, R. (2001). *Managing vulnerabilities in networked systems*. IEEE Computer.
- [19] Boehm, B., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., & Merritt, M.J. (1978). *Characteristics of Software Quality*. North-Holland.

Bibliography

- Albrecht, A. J. (1979), *Measuring application development productivity*. In *Proceedings of the Joint SHARE/GUIDE IBM Applications Development Symposium.*, IBM
- Ben-Menachem, M.; Marliss (1997), *Software Quality, Producing Practical and Consistent Software*, Thomson Computer Press
- Boehm, B.; Brown, J.R.; Kaspar, H.; MacLeod, G.J.; Merritt, M.J. (1978), *Characteristics of Software Quality*, North-Holland.
- Chidamber, S.; Kemerer (1994), *A Metrics Suite for Object Oriented Design*. *IEEE Transactions on Software Engineering*, 20 (6), pp. 476-493
- Ebert, Christof; Dumke, Reiner, *Software Measurement: Establish - Extract - Evaluate - Execute*, Kindle Edition, p. 91
- Garmus, D.; Herron (2001), *Function Point Analysis*, Addison Wesley
- Halstead, M.E. (1977), *Elements of Software Science*, Elsevier North-Holland
- Hamill, M.; Goseva-Popstojanova, K. (2009), *Common faults in software fault and failure data*. *IEEE Transactions of Software Engineering*, 35 (4), pp. 484-496.
- Jackson, D.J. (2009), *A direct path to dependable software*. *Communications of the ACM*, 52 (4).
- Martin, R. (2001)), *Managing vulnerabilities in networked systems*, IEEE Computer.
- McCabe, T. (December 1976.), *A complexity measure*. *IEEE Transactions on Software Engineering*
- McConnell, Steve (1993), *Code Complete* (First ed.), Microsoft Press
- Nygard, M.T. (2007)), *Release It! Design and Deploy Production Ready Software*, The Pragmatic Programmers.
- Park, R.E. (1992), *Software Size Measurement: A Framework for Counting Source Statements*. (CMU/SEI-92-TR-020)., Software Engineering Institute, Carnegie Mellon University
- Pressman, Scott (2005), *Software Engineering: A Practitioner's Approach* (Sixth, International ed.), McGraw-Hill Education
- Spinellis, D. (2006), *Code Quality*, Addison Wesley

Further reading

- International Organization for Standardization. *Software Engineering—Product Quality—Part 1: Quality Model*. ISO, Geneva, Switzerland, 2001. ISO/IEC 9126-1:2001(E).
- Diomidis Spinellis. *Code Quality: The Open Source Perspective* (<http://www.spinellis.gr/codequality>). Addison Wesley, Boston, MA, 2006.
- Ho-Won Jung, Seung-Gweon Kim, and Chang-Sin Chung. Measuring software product quality: A survey of ISO/IEC 9126 (<http://doi.ieeeecomputersociety.org/10.1109/MS.2004.1331309>). *IEEE Software*, 21(5):10–13, September/October 2004.
- Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, MA, second edition, 2002.
- Omar Alshathry, Helge Janicke, "Optimizing Software Quality Assurance," compsocw, pp. 87–92, 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, 2010.
- Robert L. Glass. *Building Quality Software*. Prentice Hall, Upper Saddle River, NJ, 1992.

- Roland Petrasch, "The Definition of, Software Quality': A Practical Approach (<http://www.chillarege.com/fastabstracts/issre99/99124.pdf>)", ISSRE, 1999

External links

- Linux: Fewer Bugs Than Rivals (<http://www.wired.com/software/coolapps/news/2004/12/66022>) Wired Magazine, 2004

Software quality

In the context of software engineering, software quality refers to **two related but distinct notions** that exist wherever quality is defined in a business context:

- **Software functional quality** reflects how well it complies with or conforms to a given design, based on functional requirements or specifications. That attribute can also be described as the **fitness for purpose** of a piece of software or how it compares to competitors in the marketplace as a worthwhile product^[1];
- **Software structural quality** refers to how it meets non-functional requirements that support the delivery of the functional requirements, such as robustness or maintainability, the degree to which the software was produced correctly.

Structural quality is evaluated through the analysis of the software inner structure, its source code, in effect how its architecture adheres to sound principles of software architecture. In contrast, **functional quality** is typically enforced and measured through software testing.

Historically, the structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the **ISO 9126-3** and the subsequent **25000:2005**^[2] quality model. Based on these models, the software structural quality characteristics have been clearly defined by the **Consortium for IT Software Quality (CISQ)**, an independent organization founded by the **Software Engineering Institute (SEI)**^[3] at Carnegie Mellon University, and the **Object Management Group (OMG)**^[4].

CISQ has defined 5 major desirable characteristics needed for a piece of software to provide business value: **Reliability, Efficiency, Security, Maintainability** and (adequate) **Size**.

Software quality measurement is about quantifying to what extent a software or system rates along each of these five dimensions. An aggregated measure of software quality can be computed through a qualitative or a quantitative scoring scheme or a mix of both and then a weighting system reflecting the priorities. This view of software quality being positioned on a linear continuum has to be supplemented by the analysis of **Critical Programming Errors** that under specific circumstances can lead to catastrophic outages or performance degradations that make a given system unsuitable for use regardless of rating based on aggregated measurements.

Motivation for Defining Software Quality

"A science is as mature as its measurement tools," (Louis Pasteur in Ebert and al.l, p. 91) and software engineering has evolved to a level of maturity that makes it not only possible but also necessary to measure quality software for at least two reasons:

- **Risk Management:** Software failure has caused more than inconvenience. Software errors have caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper^[2]. This resulted in requirements for the development of some types of software, particularly and historically for software embedded in medical and other devices that regulate critical infrastructures: "[Engineers who write embedded software] see Java programs stalling for one third of a second to perform garbage collection and update the user interface, and

they envision airplanes falling out of the sky."^[3] In the United States, within the Federal Aviation Administration (FAA), the Aircraft Certification Service^[7] provides software programs, policy, guidance and training, focus on software and Complex Electronic Hardware that has an effect on the airborne product (a "product" is an aircraft, an engine, or a propeller)".

- **Cost Management:** As in any other fields of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs. Industry data demonstrate that poor application structural quality in core business applications (such as Enterprise Resource Planning (ERP), Customer Relationship Management (CRM) or large transaction processing systems in financial services) results in cost and schedule overruns and creates waste in the form of rework (up to 45% of development time in some organizations^[4]). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems.

However, the distinction between measuring and improving **software quality in an embedded system** (with emphasis on risk management) and **software quality in business software** (with emphasis on cost and maintainability management) is becoming somewhat irrelevant. Embedded systems now often include a user interface and their designers are as much concerned with issues affecting usability and user productivity as their counterparts who focus on business applications. The latter are in turn looking at ERP or CRM system as a corporate nervous system whose uptime and performance are vital to the well-being of the enterprise. This convergence is most visible in mobile computing: a user who accesses an ERP application on their smartphone is depending on the quality of software across all types of software layers.

Both types of software now use multi-layered technology stacks and complex architecture so software quality analysis and measurement have to be managed in a comprehensive and consistent manner, decoupled from the software's ultimate purpose or use. In both cases, engineers and management need to be able to make rational decisions based on measurement and fact-based analysis in adherence to the precept "*In God (we) trust. All others bring data*". ((mis-)attributed to W. Edwards Deming and others).

Definition

Even though (as noted in the article on quality in business) "quality is a perceptual, conditional and somewhat subjective attribute and may be understood differently by different people," **Software structural quality** characteristics have been clearly defined by the **Consortium for IT Software Quality (CISQ)**, an independent organization founded by the Software Engineering Institute (SEI) at Carnegie Mellon University (<http://www.sei.cmu.edu>), and the Object Management Group (OMG) (<http://www.omg.org>). Under the guidance of Bill Curtis, co-author of the Capability Maturity Model framework and CISQ's first Director and Capers Jones, CISQ's Distinguished Advisor, CISQ has defined 5 major desirable characteristics of a piece of software needed to provide business value. In the House of Quality model, these are "Whats" that need to be achieved:

- **Reliability:** An attribute of resiliency and structural solidity. Reliability measures the level of risk and the likelihood of potential application failures. It also measures the defects injected due to modifications made to the software (its "stability" as termed by ISO). The goal for checking and monitoring Reliability is to reduce and prevent application downtime, application outages and errors that directly affect users, and enhance the image of IT and its impact on a company's business performance.
- **Efficiency:** The source code and software architecture attributes are the elements that ensure high performance once the application is in run-time mode. Efficiency is especially important for applications in high execution speed environments such as algorithmic or transactional processing where performance and scalability are paramount. An analysis of source code efficiency and scalability provides a clear picture of the latent business risks and the harm they can cause to customer satisfaction due to response-time degradation.

- **Security:** A measure of the likelihood of potential security breaches due to poor coding and architectural practices. This quantifies the risk of encountering critical vulnerabilities that damage the business.
- **Maintainability:** Maintainability includes the notion of adaptability, portability and transferability (from one development team to another). Measuring and monitoring maintainability is a must for mission-critical applications where change is driven by tight time-to-market schedules and where it is important for IT to remain responsive to business-driven changes. It is also essential to keep maintenance costs under control.
- **Size:** While not a quality attribute per se, the sizing of source code is a software characteristic that obviously impacts maintainability. Combined with the above quality characteristics, software size can be used to assess the amount of work produced and to be done by teams, as well as their productivity through correlation with time-sheet data, and other SDLC-related metrics.

Software functional quality is defined as conformance to explicitly stated functional requirements, identified for example using **Voice of the Customer** analysis (part of the **Design for Six Sigma** toolkit and/or documented through **use cases**) and the level of satisfaction experienced by end-users. The later is referred as to as **usability** and is concerned with how intuitive and responsive the **user interface** is, how easy simple and complex operations can be performed, how useful **error messages** are. Typically, software testing practices and tools insure that a piece of software behaves in compliance with the original design, planned user experience and desired testability, ie a software's disposition to support acceptance criteria.

The dual structural/functional dimension of software quality is consistent with the model proposed in **Steve McConnell's Code Complete** which divides software characteristics into two pieces: internal and external quality characteristics. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not ^[5].

Alternative Approaches to Software Quality Definition

One of the challenges in defining quality is that *"everyone feels they understand it"* ^[6] and other definitions of software quality could be based on extending the various description of the concept of quality used in business (see a list of possible definition here.)

Dr. **Tom DeMarco** has proposed that *"a product's quality is a function of how much it changes the world for the better."* ^[7]. This can be interpreted as meaning that functional quality and user satisfaction, is more important than structural quality in determining software quality.

Another definition, coined by **Gerald Weinberg** in *Quality Software Management: Systems Thinking*, is *"Quality is value to some person."* This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as *"Who are the people we want to value our software?"* and *"What will be valuable to them?"*

Software Quality Measurement

Although the concepts presented in this section are applicable to both Software Structural and Functional Quality, measurement of the latter is essentially performed through testing, see main article: Software Testing.

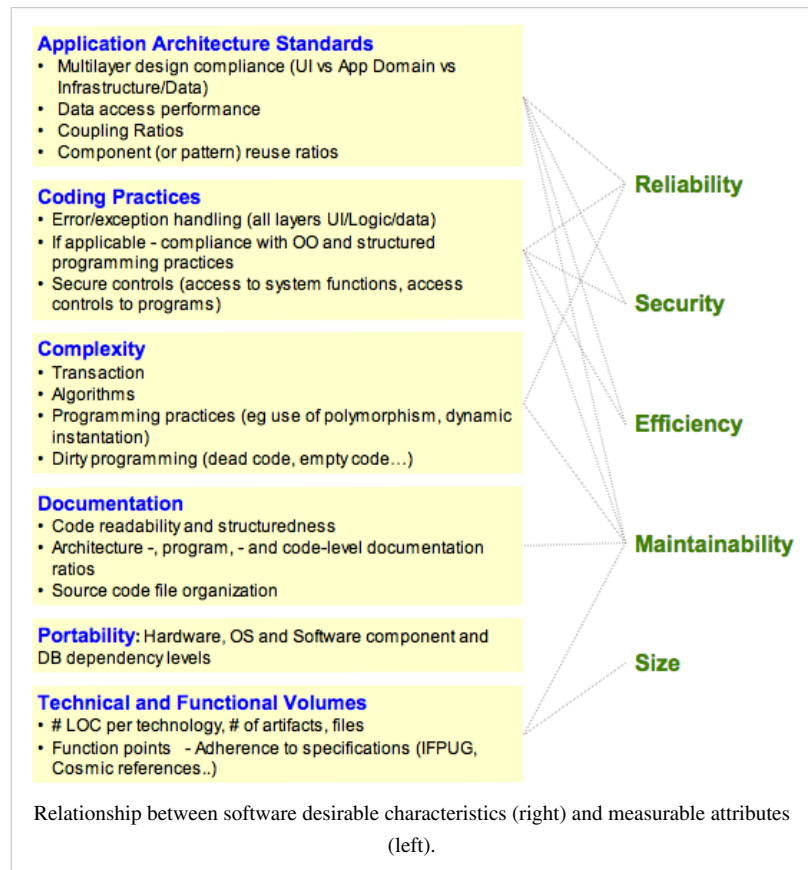
Introduction

Software quality measurement is about quantifying to what extent a software or system possesses desirable characteristics. This can be performed through qualitative or quantitative means or a mix of both. In both cases, for each **desirable characteristic**, there are a set of **measurable attributes** the existence of which in a piece of software or system tend to be correlated and associated to this characteristic. For example, an attribute associated with portability is the number of target-dependent statements in a program. More precisely, using the **Quality Function Deployment** approach, these measurable attributes are the "Hows" that need to be enforced to enable the "whats" in the Software Quality definition above.

The structure, classification and terminology of attributes and metrics applicable to software quality management have been derived or extracted from the **ISO 9126-3** and the subsequent **ISO 25000:2005** ^[8] quality model. The main focus is on internal structural quality. Subcategories have been created to handle specific areas like business application architecture and technical characteristics such as data access and manipulation or the notion of transactions.

The dependence tree between software quality characteristics and their measurable attributes is represented in the diagram on the right, where each of the 5 characteristics that matter for the user (right) or owner of the business system depends on measurable attributes (left):

- Application Architecture Practices
- Coding Practices
- Application Complexity
- Documentation
- Portability
- Technical & Functional Volume



Code-Based Analysis of Software Quality Attributes

Many of the existing software measures count structural elements of the application that result from parsing the source code such as individual instructions (Park, 1992)^[9], tokens (Halstead, 1977)^[10], control structures (McCabe, 1976), and objects (Chidamber & Kemerer, 1994)^[11].

Software quality measurement is about quantifying to what extent a software or system rate along these dimensions. The analysis can be performed using a qualitative, quantitative approach or a mix of both to provide an aggregate view (using for example weighted average(s) that reflect relative importance between the factor being measured).

This view of software quality on a linear continuum has to be supplemented by the identification of **discrete Critical Programming Errors**. These vulnerabilities may not fail a test case, but they are the result of bad practices that under specific circumstances can lead to catastrophic outages, performance degradations, security breaches, corrupted data, and myriad other problems (Nygard, 2007)^[12] that makes a given system de facto unsuitable for use regardless of its rating based on aggregated measurements. A well known example of vulnerability is the **Common Weakness Enumeration** at <http://cwe.mitre.org/> (Martin, 2001)^[13], a repository of vulnerabilities in the source code that make applications exposed to security breaches.

The measurement of critical application characteristics involves measuring structural attributes of the application's architecture, coding, in-line documentation, as displayed in the picture above. Thus, each characteristic is affected by attributes at numerous levels of abstraction in the application and all of which must be included calculating the characteristic's measure if it is to be a valuable predictor of quality outcomes that affect the business. The **layered approach** to calculating characteristic measures displayed in the figure above was first proposed by Boehm and his colleagues at TRW (Boehm, 1978)^[14] and is the approach taken in the ISO 9126 and 25000 series standards. These attributes can be measured from the parsed results of a static analysis of the application source code. Even dynamic characteristics of applications such as reliability and performance efficiency have their causal roots in the static structure of the application.

Structural quality analysis and measurement is performed through the analysis of the source code, the architecture, software framework, database schema in relationship to principles and standards that together define the conceptual and logical architecture of a system. This is distinct from the basic, local, component-level code analysis typically performed by development tools which are mostly concerned with implementation considerations and are crucial during debugging and testing activities.

Measuring Reliability

The root causes of poor reliability are found in a combination of non-compliance with good architectural and coding practices. This non-compliance can be detected by measuring the static quality attributes of an application. Assessing the static attributes underlying an application's reliability provides an estimate of the level of business risk and the likelihood of potential application failures and defects the application will experience when placed in operation.

Assessing reliability requires checks of at least the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Coding Practices
- Complexity of algorithms
- Complexity of programming practices
- Compliance with Object-Oriented and Structured Programming best practices (when applicable)
- Component or pattern re-use ratio
- Dirty programming
- Error & Exception handling (for all layers - GUI, Logic & Data)
- Multi-layer design compliance
- Resource bounds management
- Software avoids patterns that will lead to unexpected behaviors
- Software manages data integrity and consistency
- Transaction complexity level

Depending on the application architecture and the third-party components used (such as external libraries or frameworks), custom checks should be defined along the lines drawn by the above list of best practices to ensure a better assessment of the reliability of the delivered software.

Measuring Efficiency

As with Reliability, the causes of performance inefficiency are often found in violations of good architectural and coding practice which can be detected by measuring the static quality attributes of an application. These static attributes predict potential operational performance bottlenecks and future scalability problems, especially for applications requiring high execution speed for handling complex algorithms or huge volumes of data.

Assessing performance efficiency requires checking at least the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Appropriate interactions with expensive and/or remote resources
- Data access performance and data management
- Memory, network and disk space management
- Coding Practices
- Compliance with Object-Oriented and Structured Programming best practices (as appropriate)
- Compliance with SQL programming best practices

Measuring Security

Most security vulnerabilities result from poor coding and architectural practices such as SQL injection or cross-site scripting. These are well documented in lists maintained by CWE <http://cwe.mitre.org/> (see below), and the SEI/Computer Emergency Center (CERT) at Carnegie Mellon University.

Assessing security requires at least checking the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Multi-layer design compliance
- Security best practices (Input Validation, SQL Injection, Cross-Site Scripting, etc. See CWE's Top 25 <http://www.sans.org/top25-programming-errors/>)
- Programming Practices (code level)
- Error & Exception handling
- Security best practices (system functions access, access control to programs)

Measuring Maintainability

Maintainability includes concepts of modularity, understandability, changeability, testability, reusability, and transferability from one development team to another. These do not take the form of critical issues at the code level. Rather, poor maintainability is typically the result of thousands of minor violations with best practices in documentation, complexity avoidance strategy, and basic programming practices that make the difference between clean and easy-to-read code vs. unorganized and difficult-to-read code.

Assessing maintainability requires checking the following software engineering best practices and technical attributes:

- Application Architecture Practices
- Architecture, Programs and Code documentation embedded in source code
- Code readability
- Complexity level of transactions
- Complexity of algorithms
- Complexity of programming practices
- Compliance with Object-Oriented and Structured Programming best practices (when applicable)
- Component or pattern re-use ratio
- Controlled level of dynamic coding
- Coupling ratio
- Dirty programming
- Documentation
- Hardware, OS, middleware, software components and database independence
- Multi-layer design compliance
- Portability
- Programming Practices (code level)
- Reduced duplicated code and functions
- Source code file organization cleanliness

Measuring Size

Measuring software size requires that the whole source code be correctly gathered, including database structure scripts, data manipulation source code, component headers, configuration files etc. There are essentially two types of software sizes to be measured, the **technical size (footprint)** and the **functional size**:

- There are several software technical sizing methods that have been widely described here: http://en.wikipedia.org/wiki/Software_Sizing. The most common technical sizing method is number of Lines Of Code (#LOC) per technology, number of files, functions, classes, tables, etc, from which backfiring Function Points can be computed;
- The most common for measuring functional size is **Function Point Analysis** http://en.wikipedia.org/wiki/Function_point#See_also. Function Point Analysis measures the size of the software deliverable from a user's perspective. Function Point sizing is done based on user requirements and provides an accurate representation of both size for the developer/estimator and value (functionality to be delivered) and reflects the business functionality being delivered to the customer. The method includes the identification and weighting of user recognizable inputs, outputs and data stores. The size value is then available for use in conjunction with numerous measures to quantify and to evaluate software delivery and performance (Development Cost per Function Point; Delivered Defects per Function Point; Function Points per Staff Month..).

The Function Point Analysis sizing standard is supported by the **International Function Point Users Group (IFPUG)** (www.ifpug.org). It can be applied early in the software development life-cycle and it is not dependent on lines of code like the somewhat inaccurate Backfiring method. The method is technology agnostic and can be used for comparative analysis across organizations and across industries.

Since the inception of Function Point Analysis, several variations have evolved and the family of functional sizing techniques has broadened to include such sizing measures as COSMIC , NESMA, Use Case Points, FP Lite, Early and Quick FPs, and most recently Story Points. However, Function Points has a history of statistical accuracy, and has been used as a common unit of work measurement in numerous application development management (ADM) or outsourcing engagements, serving as the 'currency' by which services are delivered and performance is measured.

One common limitation to the Function Point methodology is that it is a manual process and therefore it can be labor intensive and costly in large scale initiatives such as application development or outsourcing engagements. This negative aspect of applying the methodology may be what motivated industry IT leaders to form the Consortium for IT Software Quality (www.it-cisq.org) focused on introducing a computable metrics standard for automating the measuring of software size while the IFPUG www.ifpug.org keep promoting a manual approach as most of its activity rely on FP counters certifications.

Identifying Critical Programming Errors

Critical Programming Errors are specific architectural and/or coding bad practices that result in the highest, immediate or long term, business disruption risk.

These are quite often technology-related and depend heavily on the context, business objectives and risks. Some may consider respect for naming conventions while others – those preparing the ground for a knowledge transfer for example – will consider it as absolutely critical.

Critical Programming Errors can also be classified per CISQ Characteristics. Basic example below:

- Reliability
 - Avoid software patterns that will lead to unexpected behavior (Uninitialized variable, null pointers, etc.)
 - Methods, procedures and functions doing Insert, Update, Delete, Create Table or Select must include error management
 - Multi-thread functions should be made thread safe, for instance servlets or strutsstruts action classes must not have instance/non-final static fields
- Efficiency
 - Ensure centralization of client requests (incoming and data) to reduce network traffic
 - Avoid SQL queries that don't use an index against large tables in a loop
- Security
 - Avoid fields in servlet classes that are not final static
 - Avoid data access without including error management
 - Check control return codes and implement error handling mechanisms
 - Ensure input validation to avoid cross-site scripting flaws or SQL injections flaws
- Maintainability
 - Deep inheritance trees and nesting should be avoided to improve comprehensibility
 - Modules should be loosely coupled (fanout, intermediaries,) to avoid propagation of modifications
 - Enforce homogeneous naming conventions

References

Notes

- [1] Pressman, p. 388
- [2] <http://sunnyday.mit.edu/papers/therac.pdf> (PDF)
- [3] <http://ptolemy.eecs.berkeley.edu/publications/papers/02/embsoft/embsoftwre.pdf>
- [4] <http://www.ralphyoung.net/conferences/ImprovingQuality2.ppt>
- [5] DeMarco, T., *Management Can Make Quality (Im)possible*, Cutter IT Summit, Boston, April 1999
- [6] Crosby, P., *Quality is Free*, McGraw-Hill, 1979
- [7] http://en.wikipedia.org/w/index.php?title=Software_quality&diff=436729813&oldid=434635568#CITEREFMcConnell1993
- [8] http://webstore.iec.ch/preview/info_isoiec25000%7Bed1.0%7Den.pdf
- [9] Park, R.E. (1992). Software Size Measurement: A Framework for Counting Source Statements. (CMU/SEI-92-TR-020). Software Engineering Institute, Carnegie Mellon University
- [10] Halstead, M.E. (1977). Elements of Software Science. Elsevier North-Holland.

- [11] Chidamber, S. & C. Kemerer. C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20 (6), 476-493
- [12] Nygard, M.T. (2007). *Release It! Design and Deploy Production Ready Software*. The Pragmatic Programmers.
- [13] Martin, R. (2001). *Managing vulnerabilities in networked systems*. IEEE Computer.
- [14] Boehm, B., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., & Merritt, M.J. (1978). *Characteristics of Software Quality*. North-Holland.

Bibliography

- Albrecht, A. J. (1979), *Measuring application development productivity*. In *Proceedings of the Joint SHARE/GUIDE IBM Applications Development Symposium.*, IBM
- Ben-Menachem, M.; Marliss (1997), *Software Quality, Producing Practical and Consistent Software*, Thomson Computer Press
- Boehm, B.; Brown, J.R.; Kaspar, H.; MacLeod, G.J.; Merritt, M.J. (1978), *Characteristics of Software Quality*, North-Holland.
- Chidamber, S.; Kemerer (1994), *A Metrics Suite for Object Oriented Design*. *IEEE Transactions on Software Engineering*, 20 (6), pp. 476-493
- Ebert, Christof; Dumke, Reiner, *Software Measurement: Establish - Extract - Evaluate - Execute*, Kindle Edition, p. 91
- Garmus, D.; Herron (2001), *Function Point Analysis*, Addison Wesley
- Halstead, M.E. (1977), *Elements of Software Science*, Elsevier North-Holland
- Hamill, M.; Goseva-Popstojanova, K. (2009), *Common faults in software fault and failure data*. *IEEE Transactions of Software Engineering*, 35 (4), pp. 484-496.
- Jackson, D.J. (2009), *A direct path to dependable software*. *Communications of the ACM*, 52 (4).
- Martin, R. (2001)), *Managing vulnerabilities in networked systems*, IEEE Computer.
- McCabe, T. (December 1976.), *A complexity measure*. *IEEE Transactions on Software Engineering*
- McConnell, Steve (1993), *Code Complete* (First ed.), Microsoft Press
- Nygard, M.T. (2007)), *Release It! Design and Deploy Production Ready Software*, The Pragmatic Programmers.
- Park, R.E. (1992), *Software Size Measurement: A Framework for Counting Source Statements*. (CMU/SEI-92-TR-020)., Software Engineering Institute, Carnegie Mellon University
- Pressman, Scott (2005), *Software Engineering: A Practitioner's Approach* (Sixth, International ed.), McGraw-Hill Education
- Spinellis, D. (2006), *Code Quality*, Addison Wesley

Further reading

- International Organization for Standardization. *Software Engineering—Product Quality—Part 1: Quality Model*. ISO, Geneva, Switzerland, 2001. ISO/IEC 9126-1:2001(E).
- Diomidis Spinellis. *Code Quality: The Open Source Perspective* (<http://www.spinellis.gr/codequality>). Addison Wesley, Boston, MA, 2006.
- Ho-Won Jung, Seung-Gweon Kim, and Chang-Sin Chung. Measuring software product quality: A survey of ISO/IEC 9126 (<http://doi.ieeecomputersociety.org/10.1109/MS.2004.1331309>). *IEEE Software*, 21(5):10–13, September/October 2004.
- Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, MA, second edition, 2002.
- Omar Alshathry, Helge Janicke, "Optimizing Software Quality Assurance," *compsacw*, pp. 87–92, 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, 2010.
- Robert L. Glass. *Building Quality Software*. Prentice Hall, Upper Saddle River, NJ, 1992.
- Roland Petrasch, "The Definition of, Software Quality': A Practical Approach (<http://www.chillarege.com/fastabstracts/issre99/99124.pdf>)", ISSRE, 1999

External links

- Linux: Fewer Bugs Than Rivals (<http://www.wired.com/software/coolapps/news/2004/12/66022>) Wired Magazine, 2004

Recovery testing

In software testing, **recovery testing** is the activity of testing how well an application is able to recover from crashes, hardware failures and other similar problems.

Recovery testing is the forced failure of the software in a variety of ways to verify that recovery is properly performed. Recovery testing should not be confused with reliability testing, which tries to discover the specific point at which failure occurs. Recovery testing is basically done in order to check how fast and better the application can recover against any type of crash or hardware failure etc. Type or extent of recovery is specified in the requirement specifications. It is basically testing how well a system recovers from crashes, hardware failures, or other catastrophic problems

Examples of recovery testing:

1. While an application is running, suddenly restart the computer, and afterwards check the validness of the application's data integrity.
2. While an application is receiving data from a network, unplug the connecting cable. After some time, plug the cable back in and analyze the application's ability to continue receiving data from the point at which the network connection disappeared.
3. Restart the system while a browser has a definite number of sessions. Afterwards, check that the browser is able to recover all of them.

Regression testing

Regression testing is any type of software testing that seeks to uncover new errors, or *regressions*, in existing functionality after changes have been made to a system, such as functional enhancements, patches or configuration changes.

The intent of regression testing is to ensure that a change, such as a bugfix, did not introduce new faults.^[1] "One of the main reasons for regression testing is that it's often extremely difficult for a programmer to figure out how a change in one part of the software will echo in other parts of the software."^[2]

Common methods of regression testing include rerunning previously run tests and checking whether program behavior has changed and whether previously fixed faults have re-emerged. Regression testing can be used to test a system efficiently by systematically selecting the appropriate minimum set of tests needed to adequately cover a particular change.

Background

Experience has shown that as software is fixed, emergence of new and/or reemergence of old faults is quite common. Sometimes reemergence occurs because a fix gets lost through poor revision control practices (or simple human error in revision control). Often, a fix for a problem will be "fragile" in that it fixes the problem in the narrow case where it was first observed but not in more general cases which may arise over the lifetime of the software. Frequently, a fix for a problem in one area inadvertently causes a software bug in another area. Finally, often when some feature is redesigned, some of the same mistakes that were made in the original implementation of the feature were made in the redesign.

Therefore, in most software development situations it is considered good coding practice that when a bug is located and fixed, a test that exposes the bug is recorded and regularly retested after subsequent changes to the program.^[3] Although this may be done through manual testing procedures using programming techniques, it is often done using automated testing tools.^[4] Such a test suite contains software tools that allow the testing environment to execute all the regression test cases automatically; some projects even set up automated systems to automatically re-run all regression tests at specified intervals and report any failures (which could imply a regression or an out-of-date test).^[5] Common strategies are to run such a system after every successful compile (for small projects), every night, or once a week. Those strategies can be automated by an external tool, such as BuildBot, Tinderbox, Hudson or Jenkins.

Regression testing is an integral part of the extreme programming software development method. In this method, design documents are replaced by extensive, repeatable, and automated testing of the entire software package throughout each stage of the software development cycle.

In the corporate world, regression testing has traditionally been performed by a software quality assurance team after the development team has completed work. However, defects found at this stage are the most costly to fix. This problem is being addressed by the rise of unit testing. Although developers have always written test cases as part of the development cycle, these test cases have generally been either functional tests or unit tests that verify only intended outcomes. Developer testing compels a developer to focus on unit testing and to include both positive and negative test cases.^[6]

Uses

Regression testing can be used not only for testing the *correctness* of a program, but often also for tracking the quality of its output.^[7] For instance, in the design of a compiler, regression testing could track the code size, simulation time and time of the test suite cases.

Regression testing should be part of a test plan.^[8] Regression testing can be automated.

"Also as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such *regression testing* must indeed approximate this theoretical idea, and it is very costly."

— Fred Brooks, *The Mythical Man Month*, p 122

Regression tests can be broadly categorized as functional tests or unit tests. Functional tests exercise the complete program with various inputs. Unit tests exercise individual functions, subroutines, or object methods. Both functional testing tools and unit testing tools tend to be third party products that are not part of the compiler suite, and both tend to be automated. Functional tests may be a scripted series of program inputs, possibly even an automated mechanism for controlling mouse movements. Unit tests may be separate functions within the code itself, or driver layer that links to the code without altering the code being tested.

References

- [1] Myers, Glenford (2004). *The Art of Software Testing*. Wiley. ISBN 978-0471469124.
- [2] Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 386. ISBN 978-0-615-23372-7.
- [3] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. p. 73. ISBN 0470042125. .
- [4] Automate Regression Tests When Feasible (<http://safari.oreilly.com/0201794292/ch08lev1sec4>), Automated Testing: Selected Best Practices, Elfriede Dustin, Safari Books Online
- [5] daVeiga, Nada (February 2008). "Change Code Without Fear: Utilize a Regression Safety Net" (<http://www.ddj.com/development-tools/206105233;jsessionid=2HN1TRYZ4JGVAQSNLRSKH0CJUNN2JVN>). *Dr. Dobb's Journal*. .
- [6] Dudney, Bill (2004-12-08). "Developer Testing Is 'In': An interview with [[Alberto Savoia (<http://www.sys-con.com/read/47359.htm>)] and Kent Beck"] . Retrieved 2007-11-29.
- [7] Kolawa, Adam. "Regression Testing, Programmer to Programmer" (<http://www.wrox.com/WileyCDA/Section/id-291252.html>). *Wrox*. .
- [8] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. p. 269. ISBN 0470042125. .

External links

- Microsoft regression testing recommendations ([http://msdn.microsoft.com/en-us/library/aa292167\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa292167(VS.71).aspx))

Release engineering

Release engineering, frequently abbreviated as "releng", is a sub-discipline in software engineering concerned with the compilation, assembly, and delivery of source code into finished products or other software components. Associated with the software release life cycle, it is often said that release engineering is to software engineering as manufacturing is to an industrial process. While it is not the goal of release engineering to encumber software development with a process overlay, it is often seen as a sign of organizational and developmental maturity.

Modern release engineering is concerned with several aspects of software production:

Identifiability

Being able to identify all of the source, tools, environment, and other components that make up a particular release

Reproducibility

the ability to integrate source, third party components, data, and deployment externals of a software system in order to guarantee operational stability.

Consistency

the mission to provide a stable framework for development, deployment, audit and accountability for software components.

Agility

the ongoing research into what are the repercussions of modern software engineering practices on the productivity in the software cycle, i.e. continuous integration and push on green initiatives.

Release engineering is often the integration hub for more complex software development teams, sitting at the cross between development, product management, quality assurance and other engineering efforts, also known as DevOps. Release engineering teams are often cast in the role of gatekeepers (i.e. at Facebook, Google, Microsoft) for certain critical products where their judgement forms a parallel line of responsibility and authority in relation to production releases (pushes).

Frequently, tracking of changes in a configuration management system or revision control system is part of the domain of the release engineer. The responsibility for creating and applying a version numbering scheme into software—and tracking that number back to the specific source files to which it applies—often falls onto the release engineer. Producing or improving automation in software production is usually a goal of the release engineer. Gathering, tracking, and supplying all the tools that are required to develop and build a particular piece of software may be a release engineering task, in order to reliably reproduce or maintain software years after its initial release to customers.

While most software engineers, or software developers, do many or all of the above as a course of their work, in larger organizations the specialty of the release engineer can be applied to coordinate disparate source trees, projects, teams, and components. This frees the developers to implement features in the software and also frees the quality assurance engineers to more broadly and deeply test the produced software.

The release engineer may provide software, services, or both to software engineering and software quality assurance teams. The software provided may be build tools, assembly, or other reorganization scripts which take compilation output and place them into a pre-defined tree structure, and even to the authoring and creation of installers for use by test teams or by the ultimate consumer of the software. The services provided may include software build (compilation) automation, automated test integration, results reporting, and production of or preparation for software delivery systems—e.g., in the form of electronic media (CDs, DVDs) or electronic software distribution mechanisms.

Related disciplines

- Build automation
- Porting - Product Line Engineering includes porting of a software product from one platform to other.
- Software configuration management - Although release engineering is sometimes considered part of Software Configuration Management, the latter, being a tool or a process used by the Release Engineer, is actually more of a subset of the roles and responsibilities of the typical Release Engineer.
- Continuous integration
- Change management
- Release management
- Packaging & Deployment

References

Further reading

- "Software Release Methodology" by Michael E. Bays; ISBN 0-13-636564-7.
- "Software Configuration Management" by H. Ronald Berlack; ISBN 0-471-53049-2.
- "Design of a Methodology to Support Software Release Decisions" by H. Sassenburg; ISBN 90-367-2424-4.
- "Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation" by Jez Humble, David Farley; ISBN 0-321-60191-2

Retrofits in testing

Retrofitting the code means keeping the code in sync with respect to all the regions. Suppose, in your project you have something like Development, Staging, Qual and Production regions. You do some changes for your project in the development region. And this needs to be moved to Staging or Qual after some testing. In that case, you merge your changes with the source program of the respective region where it needs to be elevated. So, you pull your changes alone from Dev1 to Stag or Qual without any other project changes. Hope this helps.

Reverse semantic traceability

Reverse Semantic Traceability (RST) is a quality control method for verification improvement that helps to insure high quality of artifacts by backward translation at each stage of the software development process.

Brief introduction

Each stage of development process can be treated as a series of “translations” from one language to another. At the very beginning a project team deals with customer’s requirements and expectations expressed in natural language. These customer requirements sometimes might be incomplete, vague or even contradictory to each other. The first step is specification and formalization of customer expectations, transition (“translation”) of them into a formal requirement document for the future system. Then requirements are translated into system architecture and step by step the project team generates megabytes of code written in a very formal programming language. There is always a threat of inserting mistakes, misinterpreting or losing something during the translation. Even a small defect in requirement or design specifications can cause huge amounts of defects at the late stages of the project. Sometimes such misunderstandings can lead to project failure or complete customer dissatisfaction.

The highest usage scenarios of Reverse Semantic Traceability method can be:

- Validating UML models: quality engineers restore a textual description of a domain, original and restored descriptions are compared.
- Validating model changes for a new requirement: given an original and changed versions of a model, quality engineers restore the textual description of the requirement, original and restored descriptions are compared.
- Validating a bug fix: given an original and modified Source code, quality engineers restore a textual description of the bug that was fixed, original and restored descriptions are compared.
- Integrating new software engineer into a team: a new team member gets an assignment to do Reverse Semantic Traceability for the key artifacts from the current projects.

RST roles

Main roles involved in RST session are:

- authors of project artifacts (both input and output),
 - reverse engineers,
 - expert group,
 - project manager.
-

RST process



Define all project artifacts and their relationship

Reverse Semantic Traceability as a validation method can be applied to any project artifact, to any part of project artifact or even to a small piece of document or code. However, it is obvious that performing RST for all artifacts can create overhead and should be well justified (for example, for medical software where possible information loss is very critical).

It is a responsibility of company and Project manager to decide what amount of project artifacts will be “reverse engineered”. This amount depends on project specific details: trade-off matrix, project and company quality assurance policies. Also it depends on importance of particular artifact for project success and level of quality control applied to this artifact.

Amount of RST sessions for project is defined at the project planning stage.

First of all project manager should create a list of all artifacts project team will have during the project. They could be presented as a tree with dependencies and relationships. Artifacts can be present in one occurrence (like Vision document) or in several occurrences (like risks or bugs). This list can be changed later during the project but the idea behind the decisions about RST activities will be the same.

Prioritize

The second step is to analyze deliverable importance for project and level of quality control for each project artifact. Importance of document is the degree of artifact impact to project success and quality of final product. It's measured by the scale:

- Crucial (1): the quality of deliverable is very important for overall quality of project and even for project success. Examples: Functional requirements, System architecture, critical bug fixes (show stoppers), risks with high probability and critical impact.
- High (2): the deliverable has an impact to quality of final product. Examples: Test cases, User interface requirements, major severity bug fixes, risks with high expose.
- Medium (3): the artifact has a medium or indirect impact to quality of final product. Examples: Project plan, medium severity bug fixes, risks with medium expose.
- Low (4): the artifact has insignificant impact to the final product quality. Example: employees' tasks, cosmetic bugs, risks with low probability.

Level of quality control is a measure that defines amount of verification and validation activities applied to artifact, and probability of miscommunication during artifact creation.

- Low (1): No review is supposed for the artifact, miscommunication and information loss are high probable, information channel is distributed, language barrier exists etc
- Medium (2): No review is supposed for the artifact, information channel is not distributed (e.g. creator of artifact and information provider are members of one team)
- Sufficient (3): Pair development or peer review is done, information channel is not distributed.
- Excellent (4): Pair development, peer review and/or testing are done, automation or unit testing is done, or there are some tools for artifact development and validation.

Define responsible people

Success of RST session strongly depends on correct assignment of responsible people.

Perform Reverse Semantic Traceability of artifact

Reverse Semantic Traceability starts when decision that RST should be performed is made and resources for it are available.

Project manager defines what documents will be an input for RST session. For example, it can be not only an artifact to restore but some background project information. It is recommended to give to reverse engineers number of words in original text so that they have an idea about amount of text they should get as a result: it can be one sentence or several pages of text. Though, the restored text may not contain the same number of words as original text nevertheless the values should be comparable.

After that reverse engineers take the artifact and restore the original text from it. RST itself takes about 1 hour for one page text (750 words).

Value the level of quality and make a decision

To complete RST session, restored and original texts of artifact should be compared and quality of artifact should be assessed. Decision about artifacts rework and its amount is made based on this assessment.

For assessment a group of experts is formed. Experts should be aware of project domain and be an experienced enough to assess quality level of compared artifacts. For example, business analysts will be experts for comparison of vision statement and restored vision statement from scenario.

RST assessment criteria:

1. Restored and original texts have quite big differences in meaning and crucial information loss
2. Restored and original texts have some differences in meaning, important information loss
3. Restored and original texts have some differences in meaning, some insignificant information loss
4. Restored and original texts are very close, some insignificant information loss
5. Restored and original texts are very close, none information is lost

Each of experts gives his assessment, and then the average value is calculated. Depending on this value Project Manager makes a decision should both artifacts be corrected or one of them or rework is not required.

If the average RST quality level is in range from 1 to 2 the quality of artifact is poor and it is recommended not only rework of validated artifact to eliminate defects but corrections of original artifact to clear misunderstandings. In this case one more RST session after rework of artifacts is required. For artifacts that have more than 2 but less than 3 corrections of validated artifact to fix defects and eliminate information loss is required, however review of original artifact to find out if there any vague piece of information that cause misunderstandings is recommended. No additional RST sessions is needed. If the average mark is more than 3 but less than 4 then corrections of validated artifact to remove defects and insignificant information loss is supposed. If the mark is greater than 4 it means that artifact is of good quality and no special corrections or rework is required.

Obviously the final decision about rework of artifacts is made by project manager and should be based on analysis of reasons of differences in texts.

References

- Vladimir Pavlov and Anton Yatsenko, The Babel Experiment: An Advanced Pantomime-based Training in OOA&OOD with UML ^[1], *36th ACM Technical Symposium on Computer Science Education (SIG CSE 2005)*, St Louis (Missouri, USA).

External links

- Vladimir L. Pavlov website ^[2]
- OMG UML website ^[3]
- Wikipedia article on P-Modeling Framework
- P-Modeling Framework Whitepaper ^[4]
- P-Modeling Framework ^[5]

References

- [1] <http://portal.acm.org/citation.cfm?id=1047124.1047426>
[2] <http://www.vlpavlov.com/>
[3] <http://www.uml.org/>
[4] <http://www.intspei.com/Products/request.aspx>
[5] <http://www.intspei.com/Products/PMFramework.aspx>

Risk-based testing

Risk-based testing (RBT) is a type of software testing that prioritizes the tests of features and functions based on the risk of their failure - a function of their importance and likelihood or impact of failure.^{[1] [2] [3] [4]} In theory, since there is an infinite number of possible tests, any set of tests must be a subset of all possible tests. Test techniques such as boundary value analysis and state transition testing aim to find the areas most likely to be defective.

Types of Risks

The methods assess risks along a variety of dimensions:

Business or Operational

- High use of a subsystem, function or feature
- Criticality of a subsystem, function or feature, including the cost of failure

Technical

- Geographic distribution of development team
- Complexity of a subsystem or function

External

- Sponsor or executive preference
- Regulatory requirements

E-Business Failure-Mode Related^[5]

- Static content defects
- Web page integration defects
- Functional behavior-related failure
- Service (Availability and Performance) related failure
- Usability and Accessibility-related failure
- Security vulnerability
- Large Scale Integration failure

References

- [1] Paul Gerrard Risk-Based E-Business Testing Part 1, Risks and Test Strategy (<http://gerrardconsulting.com/papers/articles/EBTestingPart1.pdf>) (2000)
- [2] Paul Gerrard Risk-Based E-Business Testing Part 2, Test Techniques and Tools (<http://gerrardconsulting.com/papers/articles/EBTestingPart2.pdf>) (2000)
- [3] Bach, J. The Challenge of Good Enough Software (<http://www.satisfice.com/articles/gooden2.pdf>) (1995)
- [4] Bach, J. and Kaner, C. Exploratory and Risk Based Testing (<http://www.testineducation.org/a/nature.pdf>) (2004)
- [5] Gerrard, P. and Thompson, N. Risk-Based Testing E-Business (<http://www.riskbasedtesting.com>) (2002)

Examples

- VestaLabs Risk Based Test Strategy - <http://www.vesta-labs.com/services-riskbasedtest.aspx>
- Risk Based Testing Cloud based software (<http://www.kalistick.com/smarter-test-strategies.html>)

Robustness testing

Robustness testing is any quality assurance methodology focused on testing the robustness of software. Robustness testing has also been used to describe the process of verifying the robustness (i.e. correctness) of test cases in a test process.

ANSI and IEEE have defined robustness as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions.^[1]

The term "robustness testing" was first used by OUSPG and VTT researchers taking part in the PROTOS project. They used it to refer to the new type of automated model-based test generation they had invented for software security testing.^[2] Eventually the term Fuzzing (which security people use for mostly non-intelligent and random robustness testing) extended to also cover model-based robustness testing.

List of tools for robustness testing is maintained e.g. here: <http://www.protocoltesting.com/robustness.html> (link is dead)

An overview of robustness testing methods and tools can be found in the state of the art report^[3] of the AMBER research project.

References

- [1] "Standard Glossary of Software Engineering Terminology (ANSI)". The Institute of Electrical and Electronics Engineers Inc. 1991.
 - [2] Kaksonen, Rauli. 2001. A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications 448. 128 p. + app. 15 p. ISBN 951-38-5873-1 (soft back ed.) ISBN 951-38-5874-X (on-line ed.). https://www.ee.oulu.fi/research/ouspg/PROTOS_VTT2001-functional
 - [3] Aad van Moorsel, Henrique Madeira (eds.). D2.2 State of the Art in Resilience Assessment, Measurement and Benchmarking, Chapter 13 Robustness Testing, 2009. Available online: <http://www.amber-project.eu/stateart.php>
-

San Francisco depot

San Francisco depot is a mnemonic for the SFDPO software exploratory testing heuristic. SFDPO stands for Structure, Function, Data, Platform and Operations. Each of these represents a different aspect of a software product.

Structure

Structure is what the entire product is. This is its physical files, utility programs, physical materials such as user docs, specifications and design docs, etc.

Function

Function is what the product does. This is the product's features. How does it handle errors? What is its UI? How does it interface with the operating system?

Data

Data is what the product processes. What kinds of input does it process? This can be input from the user, the file system, etc. What kind of output or reports does it generate? Does it come with default data? Is any of its input sensitive to timing or sequencing?

Platform

Platform is what the product depends upon. What operating systems and related service packs, browsers, runtime libraries, plug-ins, languages and locales, etc. does it run on? Does the user need to configure the environment? Does it depend on third-party components?

Operations

Operations are scenarios in which the product will be used. Who are the application's users and what are their patterns and sequences of input? Where and how will they use it? What are the different ways a user can use the product's features?

External links

- [How Do You Spell Testing?](#) ^[1]

References

[1] <http://www.satisfice.com/articles/sfdpo.shtml>

Sandbox (software development)

A **sandbox** is a testing environment that isolates untested code changes and outright experimentation from the production environment or repository, in the context of software development including Web development and revision control. Sandboxing protects "live" servers and their data, vetted source code distributions, and other collections of code, data and/or content, proprietary or public, from changes that could be damaging (regardless of the intent of the author of those changes) to a mission-critical system or which could simply be difficult to revert. Sandboxes replicate at least the minimal functionality needed to accurately test the programs or other code under development (e.g. usage of the same environment variables as, or access to an identical database to that used by, the stable prior implementation intended to be modified; there are many other possibilities, as the specific functionality needs vary widely with the nature of the code and the application[s] for which it is intended.)

The concept of the sandbox (sometimes also called a **working directory**, a **test server** or **development server**) is typically built into revision control software such as CVS and Subversion (SVN), in which developers "check out" a *copy* of the source code tree, or a branch thereof, to examine and work on. Only after the developer has (hopefully) fully tested the code changes in their own sandbox should the changes be checked back into and merged with the repository and thereby made available to other developers or end users of the software.^[1]

By further analogy, the term "sandbox" can also be applied in computing and networking to other temporary or indefinite isolation areas, such as security sandboxes and search engine sandboxes (both of which have highly specific meanings), that prevent incoming data from affecting a "live" system (or aspects thereof) unless/until defined requirements or criteria have been met.

In web development

Sandboxes are equally common, though less formal, among web development projects that are not version-controlled as software projects; Web developers commonly call them "test servers" or "development servers". Under this variety of project management, each developer typically has an instance of the site (locally or on a different machine), which can be altered and tested at a particular hostname, directory path, or data port, though smaller projects may simply provide a common sandbox for all developers to use jointly. While application software development sandboxing focuses on protecting the developers from other developers' changes, Web development sandboxing tends to concentrate on ensuring that changes appear and function as intended before being merged into the master copy of the pages, scripts, text, etc. that are actually being served to the real, public userbase.

In web services

The term sandbox is commonly used for the development of Web services to refer to a mirrored production environment for use by external developers. Typically, a third-party developer will develop and create an application that will use a web service from the sandbox, which is used to allow third-party team to validate their code before migrating it to the production environment. Microsoft,^[2] Google,^[3] Amazon.com,^[4] PayPal,^[5] eBay,^[6] Yahoo,^[7] among others.

In wikis

Wikis also typically employ a shared sandbox model of testing, though it is intended principally for learning and outright experimentation with features rather than for testing of alterations to existing content (the wiki analog of source code). An edit preview mode is usually used instead to test specific changes made to the texts or layout of wikis pages.

References

- [1] Vivek Venugopalan, " Developer Sandbox (<http://www.sanchivi.com/cm/cvs-bestpractices/ar01s04.html>)" chapter 4, *CVS Best Practices*, The Linux Documentation Project, 2005. (See also *Google* (<http://www.google.ca/search?q=CVS+sandbox>) for numerous other examples from the *CVS FAQ*, *SourceForge*, etc.)
- [2] (<http://test.uddi.microsoft.com/default.aspx>)
- [3] (<http://www2.sandbox.google.com>)
- [4] (<http://developer.amazonwebservices.com/connect/entry.jspa?categoryID=25&externalID=775>)
- [5] (https://developer.paypal.com/en_US/pdf/PP_Sandbox_UserGuide.pdf)
- [6] (<http://sandbox.ebay.com/>)
- [7] (<http://searchmarketing.yahoo.com/developer/docs/V2/sandbox/index.php>)

Sanity testing

A **sanity test** or **sanity check** is a basic test to quickly evaluate whether a claim or the result of a calculation can possibly be true. It is a simple check to see if the produced material is rational (that the material's creator was thinking rationally, applying sanity). The point of a sanity test is to rule out certain classes of obviously false results, not to catch every possible error. A rule-of-thumb may be checked to perform the test. The advantage of a sanity test, over performing a complete or rigorous test, is speed.

In arithmetic, for example, when multiplying by 9, using the divisibility rule for 9 to verify that the sum of digits of the result is divisible by 9 is a sanity test - it will not catch *every* multiplication error, however it's a quick and simple method to discover *many* possible errors.

In computer science, a *sanity test* is a very brief run-through of the functionality of a computer program, system, calculation, or other analysis, to assure that part of the system or methodology works roughly as expected. This is often prior to a more exhaustive round of testing.

Mathematical

A sanity test can refer to various order-of-magnitude and other simple rule-of-thumb devices applied to cross-check mathematical calculations. For example:

- If one were to attempt to square 738 and calculated 53,874, a quick sanity check could show that this result cannot be true. Consider that $700 < 738$, yet $700^2 = 7^2 \times 100^2 = 490000 > 53874$. Since squaring positive numbers preserves their inequality, the result cannot be true, and so the calculated result is incorrect. The correct answer, $738^2 = 544,644$, is more than 10 times higher than 53,874, and so the result had been off by an order of magnitude.
- In multiplication, 918×155 is not 142135 since 918 is divisible by three but 142135 is not (digits add up to 16, not a multiple of three). Also, the product must end in the same digit as the product of end-digits $8 \times 5 = 40$, but 142135 does not end in "0" like "40", while the correct answer does: $918 \times 155 = 142290$. An even quicker check is that the product of even and odd numbers is even, whereas 142135 is odd.
- When talking about quantities in physics, the power output of a car cannot be 700 kJ since that is a unit of energy, not power (energy per unit time). See dimensional analysis.

Software development

In software development, the sanity test (a form of software testing which offers "quick, broad, and shallow testing"^[1]) determines whether it is reasonable to proceed with further testing.

Software sanity tests are commonly conflated with smoke tests.^[2] A smoke test determines whether it is *possible* to continue testing, as opposed to whether it is *reasonable*. A software smoke test determines whether the program launches and whether its interfaces are accessible and responsive (for example, the responsiveness of a web page or an input button). If the smoke test fails, it is impossible to conduct a sanity test. In contrast, the ideal sanity test exercises the smallest subset of application functions needed to determine whether the application logic is generally functional and correct (for example, an interest rate calculation for a financial application). If the sanity test fails, it is not reasonable to attempt more rigorous testing. Both sanity tests and smoke tests are ways to avoid wasting time and effort by quickly determining whether an application is too flawed to merit any rigorous testing. Many companies run sanity tests and unit tests on an automated build as part of their development process.^[3]

Sanity testing may be a tool used while manually debugging software. An overall piece of software likely involves multiple subsystems between the input and the output. When the overall system is not working as expected, a sanity test can be used to make the decision on what to test next. If one subsystem is not giving the expected result, the other subsystems can be eliminated from further investigation until the problem with this one is solved.

The Hello world program is often used as a sanity test for a development environment. If Hello World fails to compile or execute, the supporting environment likely has a configuration problem. If it works, the problem being diagnosed likely lies in the real application being diagnosed.

Another, possibly more common usage of 'sanity test' is to denote checks which are performed *within* program code, usually on arguments to functions or returns therefrom, to see if the answers can be assumed to be correct. The more complicated the routine, the more important that its response be checked. The trivial case is checking to see that a file opened, written to, or closed, did not fail on these activities – which is a sanity check often ignored by programmers. But more complex items can also be sanity-checked for various reasons.

Examples of this include bank account management systems which check that withdrawals are sane in not requesting more than the account contains, and that deposits or purchases are sane in fitting in with patterns established by historical data – large deposits may be more closely scrutinized for accuracy, large purchase transactions may be double-checked with a card holder for validity against fraud, ATM withdrawals in foreign locations never before visited by the card holder might be cleared up with him, etc.; these are "runtime" sanity checks, as opposed to the "development" sanity checks mentioned above.

References

- [1] M. A. Fecko and C. M. Lott, "Lessons learned from automating tests for an operations support system," (<http://www.chris-lott.org/work/pubs/2002-spe.pdf>) *Software--Practice and Experience*, v. 32, October 2002.
- [2] Erik van Veenendaal (ED), Standard glossary of terms used in Software Testing (<http://www.istqb.org/downloads/glossary-1.1.pdf>), International Software Testing Qualification Board.
- [3] Hassan, A. E. and Zhang, K. 2006. Using Decision Trees to Predict the Certification Result of a Build (<http://portal.acm.org/citation.cfm?id=1169218.1169318&coll=&dl=ACM&type=series&idx=SERIES10803&part=series&WantType=Proceedings&title=ASE#>). In *Proceedings of the 21st IEEE/ACM international Conference on Automated Software Engineering* (September 18 – 22, 2006). Automated Software Engineering. IEEE Computer Society, Washington, DC, 189–198.

Scalability testing

Scalability Testing, part of the battery of non-functional tests, is the testing of a software application for measuring its capability to scale up or scale out ^[1] - in terms of any of its non-functional capability - be it the user load supported, the number of transactions, the data volume etc.

Performance, scalability and reliability are usually considered together by software quality analysts.

References

[1] Scalability ([http://msdn2.microsoft.com/en-us/library/aa292172\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/aa292172(VS.71).aspx))

Further reading

Designing Distributed Applications with Visual Studio .NET: Scalability ([http://msdn2.microsoft.com/en-us/library/aa292172\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/aa292172(VS.71).aspx))

Scenario testing

Scenario testing is a software testing activity that uses **scenario** tests, or simply **scenarios**, which are based on a hypothetical story to help a person think through a complex problem or system for a testing environment. The ideal scenario has five key characteristics: it is (a) a story that is (b) motivating, (c) credible, (d) complex, and (e) easy to evaluate^[1]. These tests are usually different from test cases in that test cases are single steps whereas scenarios cover a number of steps. Test suites and scenarios can be used in concert for complete system testing.

References

[1] "An Introduction to Scenario Testing" (<http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf>). Cem Kaner. . Retrieved 2009-05-07.

External links

- Introduction to Scenario Testing (<http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf>)

Security bug

A **security bug** is a software bug that benefits someone other than intended beneficiaries in the intended ways.

Security bugs introduce security vulnerabilities by compromising one or more of:

- Authentication of users and other entities
- Authorization of access rights and privileges
- Data confidentiality
- Data integrity

Security bugs need not be identified, surfaced nor exploited to qualify as such. Some exploited ones, particularly viruses, have been known to wreak global damage at massive cost.

Causes

Security bugs, like all other software bugs, stem from root causes that can generally be traced to either absent or inadequate:

- Software developer training
- Use case analysis
- Software engineering methodology
- Quality assurance testing
- ...and other best practices

Taxonomy

Security bugs generally fall into a fairly small number of broad categories that include:

- Memory safety (e.g. buffer overflow and dangling pointer bugs)
- Race condition
- Secure input and output handling
- Faulty use of an API
- Improper use case handling
- Improper exception handling
- Preprocessing input strings **after** they are checked for being acceptable.

Mitigation

See Software Security Assurance.

Security testing

Security testing is a process to determine that an information system protects data and maintains functionality as intended.

The six basic security concepts that need to be covered by security testing are: confidentiality, integrity, authentication, availability, authorization and non-repudiation. Security testing as a term has a number of different meanings and can be completed in a number of different ways. As such a Security Taxonomy helps us to understand these different approaches and meanings by providing a base level to work from.

Confidentiality

- A security measure which protects against the disclosure of information to parties other than the intended recipient that is by no means the only way of ensuring the security....

Integrity

- A measure intended to allow the receiver to determine that the information which it is providing is correct.

Authentication

This might involve confirming the identity of a person, tracing the origins of an artifact, ensuring that a product is what its packaging and labeling claims to be, or assuring that a computer program is a trusted one.

Authorization

- The process of determining that a requester is allowed to receive a service or perform an operation.
- Access control is an example of authorization.....

Availability

- Assuring information and communications services will be ready for use when expected.
- Information must be kept available to authorized persons when they need it.

Non-repudiation

- In reference to digital security, nonrepudiation means to ensure that a transferred message has been sent and received by the parties claiming to have sent and received the message. Nonrepudiation is a way to guarantee that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message.
-

Security Testing Taxonomy

Common terms used for the delivery of security testing:

- **Discovery** - The purpose of this stage is to identify systems within scope and the services in use. It is not intended to discover vulnerabilities, but version detection may highlight deprecated versions of software / firmware and thus indicate potential vulnerabilities.
 - **Vulnerability Scan** - Following the discovery stage this looks for known security issues by using automated tools to match conditions with known vulnerabilities. The reported risk level is set automatically by the tool with no manual verification or interpretation by the test vendor. This can be supplemented with credential based scanning that looks to remove some common false positives by using supplied credentials to authenticate with a service (such as local windows accounts).
 - **Vulnerability Assessment** - This uses discovery and vulnerability scanning to identify security vulnerabilities and places the findings into the context of the environment under test. An example would be removing common false positives from the report and deciding risk levels that should be applied to each report finding to improve business understanding and context.
 - **Security Assessment** - Builds upon Vulnerability Assessment by adding manual verification to confirm exposure, but does not include the exploitation of vulnerabilities to gain further access. Verification could be in the form of authorised access to a system to confirm system settings and involve examining logs, system responses, error messages, codes, etc. A Security Assessment is looking to gain a broad coverage of the systems under test but not the depth of exposure that a specific vulnerability could lead to.
 - **Penetration Test** - Penetration test simulates an attack by a malicious party. Building on the previous stages and involves exploitation of found vulnerabilities to gain further access. Using this approach will result in an understanding of the ability of an attacker to gain access to confidential information, affect data integrity or availability of a service and the respective impact. Each test is approached using a consistent and complete methodology in a way that allows the tester to use their problem solving abilities, the output from a range of tools and their own knowledge of networking and systems to find vulnerabilities that would/ could not be identified by automated tools. This approach looks at the depth of attack as compared to the Security Assessment approach that looks at the broader coverage.
 - **Security Audit** - Driven by an Audit / Risk function to look at a specific control or compliance issue. Characterised by a narrow scope, this type of engagement could make use of any of the earlier approaches discussed (vulnerability assessment, security assessment, penetration test).
 - **Security Review** - Verification that industry or internal security standards have been applied to system components or product. This is typically completed through gap analysis and utilises build / code reviews or by reviewing design documents and architecture diagrams. This activity does not utilise any of the earlier approaches (Vulnerability Assessment, Security Assessment, Penetration Test, Security Audit)
-

Semantic decision table

Semantic Decision Tables (SDT) use modern ontology engineering (OE) technologies to enhance traditional decision tables. The name "Semantic Decision Table" was coined by Yan Tang and Prof. Robert Meersman from VUB STARLab (Free University of Brussels) in 2006^[1]. An SDT is a (set of) decision table(s) properly annotated with an ontology. It provides a means to capture and examine decision makers' concepts, as well as a tool for refining their decision knowledge and facilitating knowledge sharing in a scalable manner.

Background

SDT is a decision table. A decision table is defined as a "tabular method of showing the relationship between a series of conditions and the resultant actions to be executed"^[2]. Following the de facto international standard (CSA, 1970), a decision table contains three building blocks: the conditions, the actions (or decisions), and the rules.

A *decision condition* is constructed with a *condition stub* and a *condition entry*. A *condition stub* is declared as a statement of a condition. A *condition entry* provides a value assigned to the condition stub. Similarly, an *action* (or *decision*) composes two elements: an *action stub* and an *action entry*. One states an action with an action stub. An action entry specifies whether (or in what order) the action is to be performed.

A decision table separates the data (that is the condition entries and decision/action entries) from the decision templates (that are the condition stubs, decision/action stubs, and the relations between them). Or rather, a decision table can be a tabular result of its meta-rules.

Traditional decision tables have many advantages compared to other decision support manners, such as if-then-else programming statements, decision trees and Bayesian networks. A traditional decision table is compact and easily understandable. However, it still has several limitations. For instance, a decision table often faces the problems of *conceptual ambiguity* and *conceptual duplication*; and it is *time consuming* to create and maintain *large* decision tables. Semantic Decision Tables are an attempt to solve these problems.

Definition

SDT is modeled based on the framework of Developing Ontology-Grounded Methods and Applications (DOGMA^[3]). The separation of an ontology into extremely simple linguistic structures (also known as lexons) and a layer of lexon constraints used by applications (also known as ontological commitments), aiming to achieve a degree of scalability.

According to the DOGMA framework, an SDT consists of a layer of the decision binary fact types called SDT lexons and a SDT commitment layer that consists of the constraints and axioms of these fact types.

A *lexon* l is a quintuple $\langle \gamma, t_1, r_1, r_2, t_2 \rangle$. t_1 and t_2 represent two concepts in a natural language (e.g. English); r_1 and r_2 (in, r_1 corresponds to "role" and r_2 - "co-role") refer to the relationships that the concepts share with respect to one another; γ is a context identifier refers to a context, which serves to disambiguate the terms t_1 , t_2 into the intended concepts, and in which they become meaningful.

For example, a lexon $\langle \gamma, \text{driver's license, is issued to, has, driver} \rangle$ explains a fact that "a driver's license is issued to a driver", and "a driver has a driver's license".

The ontological commitment layer formally defines selected rules and constraints by which an application (or "agent") may make use of lexons. A commitment can contain various constraints, rules and axiomatized binary facts based on needs. It can be modeled in different modeling tools, such as Object Role Modeling (ORM), Conceptual Graph, (CG) and Unified Modeling Language (UML).

SDT model

An SDT contains richer decision rules than a decision table. During the annotation process, the decision makers need to specify all the implicit rules, including the hidden decision rules and the meta-rules of (a set of) decision table(s). The semantics of these rules is derived from an agreement between the decision makers observing the real-world decision problems. The process of capturing semantics within a community is a process of knowledge acquisition.

Notes

- [1] Yan Tang and Robert Meersman (2007). C. Man-chung, J.N.K. Liu, R. Cheung, J.Zhou. ed. *Towards building semantic decision table with domain ontologies*. Proc. of International conference of information Technology and Management (ICITM2007). ISM Press. pp. 14–21. ISBN 988-97311-5-0.
- [2] Canadian Standards Association (1970). *Z243.1-1970 for Decision Tables*.
- [3] Robert Meersman (2001). d'Atri, A. & Missikoff, M.. ed. *Ontologies and Databases: More than a Fleeting Resemblance*. Proc. of OES/SEO 2001 Rome Workshop. Luiss Publication.

References

- Canadian Standards Association (1970). *Z243.1-1970 for Decision Tables*.
- Yan Tang and Robert Meersman (2007). C. Man-chung, J.N.K. Liu, R. Cheung, J.Zhou. ed. *Towards building semantic decision table with domain ontologies*. Proc. of International conference of information Technology and Management (ICITM2007). ISM Press. pp. 14–21. ISBN 988-97311-5-0.
- Yan Tang and Robert Meersman (2008). Man-Chung Chan, Ronnie Cheung & James N K Liu. ed. *Towards Building Semantic Decision Tables with Domain Ontologies*. Challenges in Information Technology Management. World Scientific. ISBN 981-281-906-1.
- Yan Tang, Robert Meersman and Jan Vanthienen. S. Bhwmich, Josef Kung, Roland Wagner. ed. *Semantic Decision Tables: Self-Organizing and Reorganizable Decision Tables*. proc. of DEXA'08 (19th International Conference on Database and Expert Systems Applications). Turin, Italy: Springer.
- Yan Tang and Robert Meersman (2009). Frode Eika Sandnes, Yan Zhang et. al.. ed. *Use Semantic Decision Tables to Improve Meaning Evolution Support Systems*. special issue of the Inderscience International Journal of Autonomous and Adaptive Communications Systems (IJAAACS). ISSN 1754-8632.
- Yan Tang and Robert Meersman (2009). *SDRule Markup Language: Towards Modelling and Interchanging Ontological Commitments for Semantic Decision Making*. Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches. IGI Publishing, USA. ISBN 1-60566-402-2.

Serenity Code Coverage

Serenity

Stable release	0.2 / January 27, 2010
Written in	Java
Operating system	Cross-platform
Type	code coverage
License	MIT license
Website	[1]
As of	January 17, 2010

Serenity is a Java code coverage, complexity, dependency, abstractness and distance metrics library with a Hudson plugin for displaying reports. Serenity uses dynamic byte code manipulation to add coverage code to class files. Trend reports are generated and rendered in the GUI.

Serenity supports Ant and Maven projects.

External links

- Serenity wiki ^[2]
- iKokoon Site ^[3]
- Hudson homepage ^[4]

References

- [1] <http://www.ikokoon.com/ikokoon/>
[2] <http://wiki.hudson-ci.org/display/HUDSON/Serenity+Plugin>
[3] <http://www.ikokoon.com/ikokoon>
[4] <http://hudson.dev.java.net/>
-

Session-based testing

Session-based testing is a software test method that aims to combine accountability and exploratory testing to provide rapid defect discovery, creative on-the-fly test design, management control and metrics reporting. The method can also be used in conjunction with Scenario testing. Session-based testing was developed in 2000 by Jonathan and James Bach.

Session-based testing can be used to introduce measurement and control to an immature test process, and can form a foundation for significant improvements in productivity and error detection. Session-based testing can offer benefits when formal requirements are not present, incomplete, or changing rapidly.

Elements of session-based testing

Charter

A charter is a goal or agenda for a test session. Charters are created by the test team prior to the start of testing, but may be added or changed at any time. Often charters are created from a specification, test plan, or by examining results from previous test sessions.

Session

An uninterrupted period of time spent testing, ideally lasting one to two hours. Each session is focused on a charter, but testers can also explore new opportunities or issues during this time. The tester creates and executes test cases based on ideas, heuristics or whatever frameworks to guide them and records their progress. This might be through the use of written notes, video capture tools or by whatever method as deemed appropriate by the tester.

Session report

The session report records the test session. Usually this includes:

- Charter.
- Area tested.
- Detailed notes on how testing was conducted.
- A list of any bugs found.
- A list of issues (open questions, product or project concerns)
- Any files the tester used or created to support their testing
- Percentage of the session spent on the charter vs investigating new opportunities.
- Percentage of the session spent on:
 - Testing - creating and executing tests.
 - Bug investigation / reporting.
 - Session setup or other non-testing activities.
- Session Start time and duration.

Debrief

A debrief is a short discussion between the manager and tester (or testers) about the session report. Jon Bach, one of the co-creators of session based test management, uses the acronym PROOF to help structure his debriefing. PROOF stands for:-

- Past. What happened during the session?
- Results. What was achieved during the session?
- Obstacles. What got in the way of good testing?
- Outlook. What still needs to be done?
- Feelings. How does the tester feel about all this?^[1]

Parsing results

With a standardized Session Report, software tools can be used to parse and store the results as aggregate data for reporting and metrics. This allows reporting on the number of sessions per area or a breakdown of time spent on testing, bug investigation, and setup / other activities.

Planning

Testers using session-based testing can adjust their testing daily to fit the needs of the project. Charters can be added or dropped over time as tests are executed and/or requirements change.

References

[1] <http://www.satisfice.com/articles/sbtm.pdf>

External links

- Session-Based Test Management Site (<http://www.satisfice.com/sbtm/>)
- How to Manage and Measure ET (http://www.quardev.com/content/whitepapers/how_measure_exploratory_testing.pdf)
- Session-Based Test Lite (http://www.quardev.com/articles/sbt_lite)
- Adventures in Session-Based Testing (<http://www.workroom-productions.com/papers/AiSBTV1.2.pdf>)
- Session-Based Test Management (<http://www.satisfice.com/articles/sbtm.pdf>)
- Applying Session-Based Testing to Medical Software (<http://www.devicelink.com/mddi/archive/03/05/003.html>)
- Sessionweb - Web application to manage SBTM including debriefing and support for statistics. (<http://code.google.com/p/sessionweb/>)
- Web application based on Session-based testing software test method (<http://sites.google.com/site/sessionbasedtester/>)

SigmatationTF

SigmatationTF is a comprehensive test automation framework developed by Sigma Resources & Technologies, Inc., to focus on the test of TCP/IP based network products.

Features

The full-IP test automation framework, SigmatationTF, can control the automation test case execution by automation test scripts. It supports network topology deployment, test execution, result collection and reporting. SigmatationTF automates reusable scripts, ensures the same test cases to be executed in identical way.

Reference

1. <http://www.sigma-rt.com/product/sigmatationtf/homepage.php> [1]
2. <http://www.linksv.com/frmPressReleaseDetail.aspx?idPress=33522&cTable=partner> [2]
3. <http://www.vmecritical.com/news/db/?9448&r=1> [3]
4. http://goliath.ecnext.com/coms2/gi_0199-7767446/Sigma-Resources-Technologies-Integrates-With.html [4]

References

- [1] <http://www.sigma-rt.com/product/sigmatationtf/homepage.php>
- [2] <http://www.linksv.com/frmPressReleaseDetail.aspx?idPress=33522&cTable=partner>
- [3] <http://www.vmecritical.com/news/db/?9448&r=1>
- [4] http://goliath.ecnext.com/coms2/gi_0199-7767446/Sigma-Resources-Technologies-Integrates-With.html

Smoke testing

Smoke testing refers to the first test made after assembly or repairs to a system, to provide some assurance that the system under test will not catastrophically fail. After a *smoke test* proves that "the pipes will not leak, the keys seal properly, the circuit will not burn, or the software will not crash outright," the system is ready for more stressful testing.

The term *smoke testing* is used in several fields, including electronics, computer software development, plumbing, woodwind repair, infectious disease control, and the entertainment industry.

History of the term

The plumbing industry started using the smoke test in 1875.^[1]

Later this usage seems to have been forgotten, and the electronics industry believes it invented the term: "The phrase *smoke test* comes from [electronic] hardware testing. You plug in a new board and turn on the power. If you see smoke coming from the board, turn off the power. You don't have to do any more testing."^[2]

Smoke testing in various industries

Electronics and electrical engineering

In electronics and electrical engineering the term *smoke test* or *power on test* is used to refer to the first time a circuit under development is attached to power, which will sometimes produce actual smoke if a design or wiring mistake has been made. Most often this smoke comes from burning resistors, which produce a unique smell familiar to many technicians. For certain circuits, overheating and burning due to circuitry that is still not properly operating can be

avoided by slowly turning up the input voltage to the unit under test by using a variable autotransformer and watching the electric current consumption. As a poor-man's "autotransformer", a properly-sized incandescent light bulb in series with the power feed can provide a similar benefit: if the unit under test has a short circuit or other overload, the bulb will light up and provide a high resistance, limiting or preventing further damage to the unit being tested.

Overloaded integrated circuits typically produce "blue smoke" (or magic smoke). "Blue smoke" is the subject of jokes among technicians who refer to it as if it were a genie in the circuit: *It's the blue smoke that makes it work—let out the blue smoke and it won't do anything.*

Software development

In computer programming and software testing, *smoke testing* is a preliminary to further testing, intended to reveal simple failures severe enough to reject a prospective software release. In this case the smoke is metaphorical. A subset of test cases that cover the most important functionality of a component or system are selected and run, to ascertain if the most crucial functions of a program work correctly.^[3] For example, a smoke test may ask basic questions like "Does the program run?", "Does it open a window?", or "Does clicking the main button do anything?" The purpose is to determine whether the application is so badly broken that further testing is unnecessary. As the book "Lessons Learned in Software Testing" puts it, "smoke tests broadly cover product features in a limited time ... if key features don't work or if key bugs haven't yet been fixed, your team won't waste further time installing or testing".^[2]

Smoke testing performed on a particular build is also known as a *build verification test*.^{[2] [3] [4]}

A daily build and smoke test is among industry best practices.^[5] Smoke testing is also done by testers before accepting a build for further testing. Microsoft claims that after code reviews, "*smoke testing* is the most cost effective method for identifying and fixing defects in software".^[6] In Microsoft's case a smoke test is the process of validating code changes before they are checked into source control.

Smoke tests can either be performed manually or using an automated tool. When automated tools are used, the tests are often initiated by the same process that generates the build itself.

Smoke tests can be broadly categorized as functional tests or unit tests. Functional tests exercise the complete program with various inputs. Unit tests exercise individual functions, subroutines, or object methods. Both functional testing tools and unit testing tools tend to be third-party products that are not part of the compiler suite. Functional tests may be a scripted series of program inputs, possibly even with an automated mechanism for controlling mouse movements. Unit tests may be separate functions within the code itself, or driver layer that links to the code without altering the code being tested.

Plumbing

In plumbing a *smoke test* forces non-toxic, artificially created smoke through waste and drain pipes under a slight pressure to find leaks.^[7] Plumes of smoke form where there are defects. This test can be performed when the plumbing is brand new, but more often it is used to find sewer gas leaks that may plague a building or an area.^[7] Any sign of smoke escaping can be considered a possible site for sewer gas to escape. Sewer gas typically has a rotten egg smell and can contain methane gas, which is explosive, or hydrogen sulfide gas, which is deadly.

Plumbing *smoke tests* are also used to find places where pipes will spill fluid,^[7] and to check sanitary sewer systems for places where ground water and storm runoff can enter. Smoke testing is particularly useful in places such as ventilated sanitary sewer systems, where completely sealing the system is not practical.

When smoke testing a sanitary sewer system it is helpful to partially block off the section of sewer to be tested. This can be done by using a sand bag on the end of a rope. The sand bag is lowered into the manhole and swung into position to partially block lines. Completely blocking the line can cause water to back up and prevent smoke from

escaping through defects. Smoke testing may not be done after rain or when ground water is unusually high as this may also prevent detection of defects.

Large downdraft fans, usually powered by gasoline engines, are placed on top of open manholes at either end of the section to be tested. If possible all lines in the manholes except for the line between the manholes are partially blocked. Smoke is created using either a smoke bomb or liquid smoke. Smoke bombs are lit and placed on a grate or in a holder on top of each fan, while liquid smoke is injected into the fan via a heating chamber. The fans create a pressure differential that forces the smoke into the sewer at a pressure just above atmospheric. With properly installed plumbing, the traps will prevent the smoke from entering the house and redirect it out the plumbing vents. Defective plumbing systems or dry traps will allow smoke to enter the inside of the house.

The area around the section being tested is searched for smoke plumes. Plumes coming from plumbing vents or the interface between the fan shroud and manhole rim are normal; however, smoke plumes outside of the manhole rim are not. Plumes are marked, usually with flags, and defects are noted using measurements from stationary landmarks like the corners of houses. The plumes or markers may also be photographed.

Woodwind instrument repair

In woodwind instrument repair, a smoke test involves plugging one end of an instrument and blowing smoke into the other to test for leaks. Escaping smoke reveals improperly seated pads and faulty joints (i.e. leaks). After this test the instrument is cleaned to remove nicotine and other deposits left by the smoke.^[8] Due to tobacco smoke being used, this test may be hazardous to the health of the technician in the long run.

Described in a repair manual written in the 1930s. *smoke testing* is considered obsolete, and is no longer used by reputable technicians. The usual alternative to smoke is to place a bright light inside the instrument then check for light appearing around pads and joints.

Automotive repair

In the same way that plumbing and woodwind instruments are tested, the vacuum systems of automobiles may be tested in order to locate difficult-to-find vacuum leaks. Artificial smoke is deliberately introduced into the system under slight pressure and any leaks are indicated by the escaping smoke. Smoke can also be used to locate difficult-to-find leaks in the fuel evaporative emissions control (EVAP) system.

Infectious disease control

In infectious disease control a *smoke test* is done to see whether a room is under negative pressure. A tube containing smoke is held near the bottom of the negative pressure room door, about two inches in front of the door. The smoke tube is held parallel to the door, and a small amount of smoke is then generated by gently squeezing the bulb. Care is taken to release the smoke from the tube slowly to ensure the velocity of the smoke from the tube does not overpower the air velocity. If the room is at negative pressure, the smoke will travel under the door and into the room. If the room is not a negative pressure, the smoke will be blown outward or will stay stationary.

Entertainment

In the entertainment industry a *smoke test* is done to ensure that theatrical smoke and fog used during a live event will not set off the smoke detectors in a venue. To *smoke test* a venue the venue is filled to the full capacity with smoke to see if there are any smoke detectors still live, or if there are any leaks of smoke from the venue sufficient to set off detectors in other parts of the venue being tested.

References

- [1] Buchan, W.P. "Institution of the Smoke-Test for Drains ([http://books.google.com/books?id=GBc1AAAAMAAJ&pg=PA75&dq="smoke+test"&hl=en&ei=nuD7TfjsH5KcgQfknJ3nCg&sa=X&oi=book_result&ct=result&resnum=5&ved=0CFAQ6AEwBA#v=onepage&q="smoke test"&f=false](http://books.google.com/books?id=GBc1AAAAMAAJ&pg=PA75&dq=)). Read before the Royal Scottish Society of Arts, 13 April 1891. Retrieved 2011-06-24.
- [2] Kaner, Bach, and Pettichord. "Lessons Learned in Software Testing". Wiley Computer Publishing, 2002, p. 95. ISBN 0-471-08112-4
- [3] Dustin, Rashka, Paul. "Automated Software Testing - Introduction, Management, and Performance". Addison-Wesley 1999, p. 43-44. ISBN 0201432870.
- [4] "How to: Configure and Run Build Verification Tests (BVTs)" ([http://msdn.microsoft.com/en-us/library/ms182465\(v=VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms182465(v=VS.80).aspx)), *MSDN Library for Visual Studio 2005*, accessed November 20, 2010.
- [5] McConnell, Steve. "Rapid Development". Microsoft Press, p. 405
- [6] "Guidelines for Smoke Testing" ([http://msdn.microsoft.com/en-us/library/ms182613\(v=VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms182613(v=VS.80).aspx)), *MSDN Library for Visual Studio 2005*, accessed November 20, 2010.
- [7] "Smoke Testing: The Diagnostic Secret Weapon" (<http://www.masterplumbers.com/plumbviews/2002/smoketesting.asp>). masterplumbers.com. .
- [8] Anthony Baines, Woodwind Instruments and their history", 1967 (page 355)

External links

- PC Mag's Definition (http://www.pcmag.com/encyclopedia_term/0,2542,t=smoke+test&i=51556,00.asp)

Soak testing

Soak testing involves testing a system with a significant load extended over a significant period of time, to discover how the system behaves under sustained use.

For example, in software testing, a system may behave exactly as expected when tested for 1 hour. However, when it is tested for 3 hours, problems such as memory leaks cause the system to fail or behave randomly.

Soak tests are used primarily to check the reaction of a subject under test under a possible simulated environment for a given duration and for a given threshold. Observations made during the soak test are used to improve the characteristics of the subject under test further.

In electronics, soak testing may involve testing a system up to or above its maximum ratings for a long period of time. Some companies may soak test a product for a period of many months, while also applying external stresses such as elevated temperatures.

This falls under stress testing.

Soapsonar

SOAPSonar is a software testing and diagnostics tool for SOAP, XML and REST based Web Services. The core focus is on functional, performance, interoperability, and security testing of service endpoints by performing client simulation and automated generation of client messages.

Features

SOAPSonar provides the following feature set:

- WSDL Parsing and Conformance Scoring
- SOAP, XML, and REST service validation
- Web Service Functional Testing with Success Rule Framework
- Web Service Performance Profiling and Concurrent Client Load Testing
- Web Service Design-Time and Run-Time WS-I Basic Profile Compliance Assessment
- Web Service Security Testing with Risk Mediation
- Web Service Policy Framework Testing
- Identity Testing

Technology

SOAPSonar supports W3C and OASIS standards for XML, XSD, SOAP, WSDL, WS-Security, SAML, MIME, DIME, MTOM, X.509, XSLT. PKI support provided for SSL X.509 Authentication, WS-Security Signatures, and WS-Security Encryption and included Windows Keystore, Java Keystore, PKCS #12, and SmartCard certificates.

External links

- CNet Distribution Site ^[1]
- SOAPSonar Site ^[2]
- OASIS ^[3] (Contains links to download specification documents)
- W3C Sites ^[4]

References

[1] <http://www.download.com>

[2] <http://www.crosschecknet.com>

[3] <http://www.oasis-open.org>

[4] <http://www.w3csites.com/>

SOASTA

SOASTA, Inc. is a privately-held^[1] American technology company that provides services to test websites and web applications.^{[2] [3]} It is the leading provider of cloud-based testing services, and created the industry's first browser-based website testing product. Website tests include load testing, performance testing, functional testing and user interface testing. SOASTA provides cloud website testing with their product CloudTest, which simulates thousands of users visiting a website simultaneously using the Amazon Elastic Compute Cloud (EC2) service.^{[4] [5]} SOASTA allows customers to use predefined tests or create customized tests to automatically test their web applications.^[6]

In the first half of 2010, SOASTA was selected by The Wall Street Journal as a "Top 50" Venture-backed company,^[7] by AlwaysOn as an OnDemand Top 100 Winner,^[8] and by the Red Herring as a Top 100 North America Tech Startup.^[9]

SOASTA was founded by Ken Gardner^[1] and is based in Mountain View, California.^[10] Tom Lounibos has been CEO^[11] since September 2006.^[12]

In September 2008, SOASTA raised USD \$6.4 M in financing from Formative Ventures, Canaan Partners and The Entrepreneur's Fund.^{[10] [13]} In December 2008, SOASTA announce an alliance with SAVVIS to provide SAVVIS customer's with SOASTA's cloud testing services.^[14]

References

- [1] Bowles, Jerry (October 19, 2006). "SOASTA Founder Talks SOA and Services" (<http://www.webpronews.com/blogtalk/2006/10/19/soasta-founder-talks-soa-and-services>). *www.webpronews.com*. . Retrieved 2009-02-12.
 - [2] King, Rachael (December 14, 2007). "Tech: Let Us Serve You" (http://www.businessweek.com/magazine/content/07_72/s0712052781921.htm). *BusinessWeek*. . Retrieved 2009-02-12.
 - [3] Krill, Paul (January 28, 2009). "Cloud computing shapes up as big trend for 2009" (http://www.infoworld.com/article/09/01/28/cloud-computing-shapes-up-as-big-trend-for-2009_1.html). *InfoWorld*. . Retrieved 2009-02-12.
 - [4] Barnitzke, Armin (July 2008). "Soasta bringt das Webtesting in die Cloud" (http://www.computerzeitung.de/articles/soasta_bringt_das_webtesting_in_die_cloud:/2008028/31570833_ha_CZ.html?thes=&tp=/themen/softwareentwicklung/). *Computer Zeitung*. . Retrieved 2009-02-12.
 - [5] Murphy, Thomas (30 June 2008). "Soasta Is First To Bring Testing to the Cloud" (http://www.gartner.com/DisplayDocument?ref=g_search&id=710608). *Gartner*. . Retrieved 2009-02-12.
 - [6] Hall, Mark (26-FEB-07). "Orchestrate SOA app tests" (http://www.accessmylibrary.com/coms2/summary_0286-29973378_ITM). *Computerworld*. . Retrieved 2009-02-12.
 - [7] "The Next Big Thing: The Top 50 Venture-Backed Companies" (http://graphicsweb.wsj.com/documents/NEXT_BIG_THING/NEXT_BIG_THING.html). *Wall Street Journal*. 09-MAR-10. . Retrieved 2010-07-22.
 - [8] "The 2010 OnDemand Top 100 Private Companies" (<http://alwayson.goingon.com/2010/OnDemand-Top-100-Competiton>). *AlwaysOn*. 22-MAY-10. . Retrieved 2010-07-22.
 - [9] "Top 100 North America Tech Startups" (<http://www.herring100.com/RHNA2010/2010winners/2010winners.html>). *Red Herring*. JUNE-10. . Retrieved 2010-07-22.
 - [10] "Cloud computing company Soasta gets \$6.4M" (<http://www.bizjournals.com/sanjose/stories/2008/09/08/daily5.html>). *bizjournals*. September 8, 2008. . Retrieved 2009-02-12.
 - [11] SOASTA. "Press Release: SOASTA Performs 500th Test in the Cloud" (<http://finance.yahoo.com/news/SOASTA-Performs-500th-Test-in-iw-14163842.html>). *Yahoo! Finance*. . Retrieved 2009-02-12.
 - [12] "Mountain View's Soasta names CEO" (<http://sanjose.bizjournals.com/sanjose/stories/2006/09/11/daily3.html>). *bizjournals*. September 11, 2006. . Retrieved 2009-02-12.
 - [13] Croll, Alistair (September 8, 2008). "SOASTA Raises \$6.4M to Test in the Cloud" (<http://gigaom.com/2008/09/08/soasta-raises-64m-to-test-in-the-cloud/>). *GigaOM*. . Retrieved 2009-02-12.
 - [14] "SOASTA Teams With Savvis to Deliver Cloud Testing Service" (<http://www.marketwatch.com/news/story/soasta-teams-savvis-deliver-cloud/story.aspx?guid={E86504B0-51BD-484B-8454-1A075E06501E}>). *MarketWatch*. . Retrieved 2009-02-12.
-

External links

- Official website (<http://http://www.soasta.com/>)
- "SOASTA Company Profile" (<http://www.crunchbase.com/company/soasta>). CrunchBase.
- Chegg.com Goes Back to School With Cloud Testing From SOASTA (<http://www.soasta.com/company/news/pr20090120.html>), Press Release
- "SOASTA Concerto: A New Approach to Automated Web Testing" (http://www.soasta.com/download/A_New_Approach_to_Automated_Web_Testing.pdf). SOASTA.

Software performance testing

In software engineering, **performance testing** is testing that is performed, to determine how fast some aspect of a system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage.

Performance testing is a subset of Performance engineering, an emerging computer science practice which strives to build performance into the design and architecture of a system, prior to the onset of actual coding effort.

Performance Testing Sub-Genres

Load Testing

Load testing is the simplest form of performance testing. A load test is usually conducted to understand the behavior of the application under a specific expected load. This load can be the expected concurrent number of users on the application performing a specific number of transactions within the set duration. This test will give out the response times of all the important business critical transactions. If the database, application server, etc. are also monitored, then this simple test can itself point towards any bottlenecks in the application software...

Stress Testing

Stress testing is normally used to understand the upper limits of capacity within the application landscape. This kind of test is done to determine the application's robustness in terms of extreme load and helps application administrators to determine if the application will perform sufficiently if the current load goes well above the expected maximum.

Endurance Testing (Soak Testing)

Endurance testing is usually done to determine if the application can sustain the continuous expected load. During endurance tests, memory utilization is monitored to detect potential leaks. Also important, but often overlooked is performance degradation. That is, to ensure that the throughput and/or response times after some long period of sustained activity are as good or better than at the beginning of the test. It essentially involves applying a significant load to a system for an extended, significant period of time. The goal is to discover how the system behaves under sustained use.

Spike Testing

Spike testing, as the name suggests is done by spiking the number of users and understanding the behavior of the application; whether performance will suffer, the application will fail, or it will be able to handle dramatic changes in load.

Configuration Testing

Configuration testing is another variation on traditional performance testing. Rather than testing for performance from the perspective of load you are testing the effects of configuration changes in the application landscape on application performance and behaviour. A common example would be experimenting with different methods of load-balancing.

Isolation Testing

Isolation testing is not unique to performance testing but a term used to describe repeating a test execution that resulted in an application problem. Often used to isolate and confirm the fault domain.

Setting performance goals

Performance testing can serve different purposes.

- It can demonstrate that the system meets performance criteria.
- It can compare two systems to find which performs better.
- Or it can measure what parts of the system or workload causes the system to perform badly.

Many performance tests are undertaken without due consideration to the setting of realistic performance goals. The first question from a business perspective should always be "why are we performance testing?". These considerations are part of the business case of the testing. Performance goals will differ depending on the application technology and purpose however they should always include some of the following:

Concurrency/Throughput

If an application identifies end-users by some form of login procedure then a concurrency goal is highly desirable. By definition this is the largest number of concurrent application users that the application is expected to support at any given moment. The work-flow of your scripted transaction may impact true application concurrency especially if the iterative part contains the Login & Logout activity

If your application has no concept of end-users then your performance goal is likely to be based on a maximum throughput or transaction rate. A common example would be casual browsing of a web site such as Wikipedia.

Server response time

This refers to the time taken for one application node to respond to the request of another. A simple example would be a HTTP 'GET' request from browser client to web server. In terms of response time this is what all load testing tools actually measure. It may be relevant to set server response time goals between all nodes of the application landscape.

Render response time

A difficult thing for load testing tools to deal with as they generally have no concept of what happens within a node apart from recognizing a period of time where there is no activity 'on the wire'. To measure render response time it is generally necessary to include functional test scripts as part of the performance test scenario which is a feature not offered by many load testing tools.

Performance specifications

It is critical to detail performance specifications (requirements) and document them in any performance test plan. Ideally, this is done during the requirements development phase of any system development project, prior to any design effort. See Performance Engineering for more details.

However, **performance testing** is frequently not performed against a specification i.e. no one will have expressed what the maximum acceptable response time for a given population of users should be. Performance testing is frequently used as part of the process of performance profile tuning. The idea is to identify the “weakest link” – there is inevitably a part of the system which, if it is made to respond faster, will result in the overall system running faster. It is sometimes a difficult task to identify which part of the system represents this critical path, and some test tools include (or can have add-ons that provide) instrumentation that runs on the server (agents) and report transaction times, database access times, network overhead, and other server monitors, which can be analyzed together with the raw performance statistics. Without such instrumentation one might have to have someone crouched over Windows Task Manager at the server to see how much CPU load the performance tests are generating (assuming a Windows system is under test).

Performance testing can be performed across the web, and even done in different parts of the country, since it is known that the response times of the internet itself vary regionally. It can also be done in-house, although routers would then need to be configured to introduce the lag what would typically occur on public networks. Loads should be introduced to the system from realistic points. For example, if 50% of a system's user base will be accessing the system via a 56K modem connection and the other half over a T1, then the load injectors (computers that simulate real users) should either inject load over the same connections (ideal) or simulate the network latency of such connections, following the same user profile.

It is always helpful to have a statement of the likely peak numbers of users that might be expected to use the system at peak times. If there can also be a statement of what constitutes the maximum allowable 95 percentile response time, then an injector configuration could be used to test whether the proposed system met that specification.

Questions to ask

Performance specifications should ask the following questions, at a minimum:

- In detail, what is the performance test scope? What subsystems, interfaces, components, etc. are in and out of scope for this test?
- For the user interfaces (UIs) involved, how many concurrent users are expected for each (specify peak vs. nominal)?
- What does the target system (hardware) look like (specify all server and network appliance configurations)?
- What is the Application Workload Mix of each application component? (for example: 20% login, 40% search, 30% item select, 10% checkout).
- What is the System Workload Mix? [Multiple workloads may be simulated in a single performance test] (for example: 30% Workload A, 20% Workload B, 50% Workload C)
- What are the time requirements for any/all back-end batch processes (specify peak vs. nominal)?

Pre-requisites for Performance Testing

A stable build of the application which must resemble the Production environment as close to possible.

The performance testing environment should not be clubbed with User acceptance testing (UAT) or development environment. This is dangerous as if an UAT or Integration test or other tests are going on in the same environment, then the results obtained from the performance testing may not be reliable. As a best practice it is always advisable to have a separate performance testing environment resembling the production environment as much as possible.

Test conditions

In performance testing, it is often crucial (and often difficult to arrange) for the test conditions to be similar to the expected actual use. This is, however, not entirely possible in actual practice. The reason is that the workloads of production systems have a random nature, and while the test workloads do their best to mimic what may happen in the production environment, it is impossible to exactly replicate this workload variability - except in the most simple system.

Loosely-coupled architectural implementations (e.g.: SOA) have created additional complexities with performance testing. Enterprise services or assets (that share a common infrastructure or platform) require coordinated performance testing (with all consumers creating production-like transaction volumes and load on shared infrastructures or platforms) to truly replicate production-like states. Due to the complexity and financial and time requirements around this activity, some organizations now employ tools that can monitor and create production-like conditions (also referred as "noise") in their performance testing environments (PTE) to understand capacity and resource requirements and verify / validate quality attributes.

Timing

It is critical to the cost performance of a new system, that performance test efforts begin at the inception of the development project and extend through to deployment. The later a performance defect is detected, the higher the cost of remediation. This is true in the case of functional testing, but even more so with performance testing, due to the end-to-end nature of its scope.

Tools

In the diagnostic case, software engineers use tools such as profilers to measure what parts of a device or software contributes most to the poor performance or to establish throughput levels (and thresholds) for maintained acceptable response time.

Myths of Performance Testing

Some of the very common myths are given below.

1. Performance Testing is done to break the system.

Stress Testing is done to understand the break point of the system. Otherwise normal load testing is generally done to understand the behavior of the application under the expected user load. Depending on other requirements, such as expectation of spike load, continued load for an extended period of time would demand spike, endurance soak or stress testing.

2. Performance Testing should only be done after the System Integration Testing

Although this is mostly the norm in the industry, performance testing can also be done while the initial development of the application is taking place. This kind of approach is known as the **Early Performance Testing**. This approach would ensure a holistic development of the application keeping the performance parameters in mind. Thus the finding of a performance bug just before the release of the application and the cost involved in rectifying the bug is reduced to a great extent.

3. Performance Testing only involves creation of scripts and any application changes would cause a simple refactoring of the scripts.

Performance Testing in itself is an evolving science in the Software Industry. Scripting itself although important, is only one of the components of the performance testing. The major challenge for any performance tester is to determine the type of tests needed to execute and analyzing the various performance counters to determine the performance bottleneck.

The other segment of the myth concerning the change in application would result only in little refactoring in the scripts is also untrue as any form of change on the UI especially in the Web protocol would entail complete re-development of the scripts from scratch. This problem becomes bigger if the protocols involved include Web Services, Siebel, Citrix, and SAP.

Technology

Performance testing technology employs one or more PCs or Unix servers to act as injectors – each emulating the presence of numbers of users and each running an automated sequence of interactions (recorded as a script, or as a series of scripts to emulate different types of user interaction) with the host whose performance is being tested. Usually, a separate PC acts as a test conductor, coordinating and gathering metrics from each of the injectors and collating performance data for reporting purposes. The usual sequence is to ramp up the load – starting with a small number of virtual users and increasing the number over a period to some maximum. The test result shows how the performance varies with the load, given as number of users vs response time. Various tools, are available to perform such tests. Tools in this category usually execute a suite of tests which will emulate real users against the system. Sometimes the results can reveal oddities, e.g., that while the average response time might be acceptable, there are outliers of a few key transactions that take considerably longer to complete – something that might be caused by inefficient database queries, pictures etc.

Performance testing can be combined with stress testing, in order to see what happens when an acceptable load is exceeded –does the system crash? How long does it take to recover if a large load is reduced? Does it fail in a way that causes collateral damage?

Analytical Performance Modeling is a method to model the behaviour of an application in a spreadsheet. The model is fed with measurements of transaction resource demands (CPU, disk I/O, LAN, WAN), weighted by the transaction-mix (business transactions per hour). The weighted transaction resource demands are added-up to obtain the hourly resource demands and divided by the hourly resource capacity to obtain the resource loads. Using the responsetime formula ($R=S/(1-U)$, R=responsetime, S=servicetime, U=load), responsetimes can be calculated and calibrated with the results of the performance tests. Analytical performance modelling allows evaluation of design options and system sizing based on actual or anticipated business usage. It is therefore much faster and cheaper than performance testing, though it requires thorough understanding of the hardware platforms.

Tasks to undertake

Tasks to perform such a test would include:

- Decide whether to use internal or external resources to perform the tests, depending on inhouse expertise (or lack thereof)
- Gather or elicit performance requirements (specifications) from users and/or business analysts
- Develop a high-level plan (or project charter), including requirements, resources, timelines and milestones
- Develop a detailed performance test plan (including detailed scenarios and test cases, workloads, environment info, etc.)
- Choose test tool(s)
- Specify test data needed and charter effort (often overlooked, but often the death of a valid performance test)
- Develop proof-of-concept scripts for each application/component under test, using chosen test tools and strategies
- Develop detailed performance test project plan, including all dependencies and associated timelines
- Install and configure injectors/controller
- Configure the test environment (ideally identical hardware to the production platform), router configuration, quiet network (we don't want results upset by other users), deployment of server instrumentation, database test sets developed, etc.

- Execute tests – probably repeatedly (iteratively) in order to see whether any unaccounted for factor might affect the results
- Analyze the results - either pass/fail, or investigation of critical path and recommendation of corrective action

Methodology

Performance Testing Web Applications Methodology

According to the Microsoft Developer Network the Performance Testing Methodology ^[1] consists of the following activities:

- **Activity 1. Identify the Test Environment.** Identify the physical test environment and the production environment as well as the tools and resources available to the test team. The physical environment includes hardware, software, and network configurations. Having a thorough understanding of the entire test environment at the outset enables more efficient test design and planning and helps you identify testing challenges early in the project. In some situations, this process must be revisited periodically throughout the project's life cycle.
- **Activity 2. Identify Performance Acceptance Criteria.** Identify the response time, throughput, and resource utilization goals and constraints. In general, response time is a user concern, throughput is a business concern, and resource utilization is a system concern. Additionally, identify project success criteria that may not be captured by those goals and constraints; for example, using performance tests to evaluate what combination of configuration settings will result in the most desirable performance characteristics.
- **Activity 3. Plan and Design Tests.** Identify key scenarios, determine variability among representative users and how to simulate that variability, define test data, and establish metrics to be collected. Consolidate this information into one or more models of system usage to be implemented, executed, and analyzed.
- **Activity 4. Configure the Test Environment.** Prepare the test environment, tools, and resources necessary to execute each strategy as features and components become available for test. Ensure that the test environment is instrumented for resource monitoring as necessary.
- **Activity 5. Implement the Test Design.** Develop the performance tests in accordance with the test design.
- **Activity 6. Execute the Test.** Run and monitor your tests. Validate the tests, test data, and results collection. Execute validated tests for analysis while monitoring the test and the test environment.
- **Activity 7. Analyze Results, Tune, and Retest.** Analyse, Consolidate and share results data. Make a tuning change and retest. Improvement or degradation? Each improvement made will return smaller improvement than the previous improvement. When do you stop? When you reach a CPU bottleneck, the choices then are either improve the code or add more CPU.

External links

- The Art of Application Performance Testing - O'Reilly ISBN 978-0-596-52066-3 ^[2] (Book)
- Performance Testing Guidance for Web Applications ^[3] (MSDN)
- Performance Testing Guidance for Web Applications ^[4] (Book)
- Performance Testing Guidance for Web Applications ^[5] (PDF)
- Performance Testing Guidance ^[6] (Online KB)
- Performance Testing Videos ^[7] (MSDN)
- Open Source Performance Testing tools ^[8]
- "User Experience, not Metrics" and "Beyond Performance Testing" ^[9]
- "Performance Testing Traps / Pitfalls" ^[10]

References

- [1] <http://msdn2.microsoft.com/en-us/library/bb924376.aspx>
- [2] <http://oreilly.com/catalog/9780596520670>
- [3] <http://msdn2.microsoft.com/en-us/library/bb924375.aspx>
- [4] <http://www.amazon.com/dp/0735625700>
- [5] <http://www.codeplex.com/PerfTestingGuide/Release/ProjectReleases.aspx?ReleaseId=6690>
- [6] <http://www.codeplex.com/PerfTesting>
- [7] <http://msdn2.microsoft.com/en-us/library/bb671346.aspx>
- [8] <http://www.opensourcetesting.org/performance.php>
- [9] <http://www.perftestplus.com/pubs.htm>
- [10] http://www.mercury-consulting-ltd.com/wp/Performance_Testing_Traps.html

Software testability

Software testability is the degree to which a software artifact (i.e. a software system, software module, requirements- or design document) supports testing in a given test context.

Testability is not an intrinsic property of a software artifact and can not be measured directly (such as software size). Instead testability is an extrinsic property which results from interdependency of the software to be tested and the test goals, test methods used, and test resources (i.e., the test context).

A lower degree of testability results in increased test effort. In extreme cases a lack of testability may hinder testing parts of the software or software requirements at all.

Background

The effort and effectiveness of software tests depends on numerous factors including:

- properties of the software requirements
- properties of the software itself (such as size, complexity and **testability**)
- properties of the test methods used
- properties of the development- and testing processes
- qualification and motivation of the persons involved in the test process

Testability of Software Components

The testability of software components (modules, classes) is determined by factors such as:

- **controllability**: The degree to which it is possible to control the state of the component under test (CUT) as required for testing.
- **observability**: The degree to which it is possible to observe (intermediate and final) test results.
- **isolateability**: The degree to which the component under test (CUT) can be tested in isolation.
- **separation of concerns**: The degree to which the component under test has a single, well defined responsibility.
- **understandability**: The degree to which the component under test is documented or self-explaining.
- **automatability**: The degree to which it is possible to automate testing of the component under test.
- **heterogeneity**: The degree to which the use of diverse technologies requires to use diverse test methods and tools in parallel.

The testability of software components can be improved by:

- Test-driven development
 - design for testability (similar to design for test in the hardware domain)
-

Testability of Requirements

Requirements need to fulfill the following criteria in order to be testable:

- **consistent**
- **complete**
- **unambiguous**
- **quantitative** (a requirement like "fast response time" can not be verified)
- **verifiable in practice** (a test is feasible not only in theory but also in practice with limited resources)

References

- Robert V. Binder: *Testing Object-Oriented Systems: Models, Patterns, and Tools*, ISBN 0201809389
- Stefan Jungmayr: *Improving testability of object-oriented systems* ^[1], ISBN 3-89825-781-9
- Wanderlei Souza: *Abstract Testability Patterns* ^[2], ISSN 1884-0760

References

[1] http://www.dissertation.de/index.php3?active_document=/FDP/sj929.pdf

[2] <http://patterns-wg.fuka.info.waseda.ac.jp/SPAQU/proceedings2009/3-P2-AbstractTestabilityPatterns.pdf>

Software testing controversies

There is considerable **variety** among **software testing** writers and consultants about what constitutes responsible software testing. Members of the "context-driven" school of testing^[1] believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation. In addition, prominent members of the community consider much of the writing about software testing to be doctrine, mythology, and folklore. Some contend that this belief directly contradicts standards such as the IEEE 829 test documentation standard, and organizations such as the Food and Drug Administration who promote them. The context-driven school's retort is that *Lessons Learned in Software Testing* includes one lesson supporting the use IEEE 829 and another opposing it; that not all software testing occurs in a regulated environment and that practices appropriate for such environments would be ruinously expensive, unnecessary, and inappropriate for other contexts; and that in any case the FDA generally promotes the principle of the least burdensome approach.

Some of the major controversies include:

Agile vs. traditional

Starting around 1990, a new style of writing about testing began to challenge what had come before. The seminal work in this regard is widely considered to be *Testing Computer Software*, by Cem Kaner.^[2] Instead of assuming that testers have full access to source code and complete specifications, these writers, including Kaner and James Bach, argued that testers must learn to work under conditions of uncertainty and constant change. Meanwhile, an opposing trend toward process "maturity" also gained ground, in the form of the Capability Maturity Model. The agile testing movement (which includes but is not limited to forms of testing practiced on agile development projects) has popularity mainly in commercial circles, whereas the CMM was embraced by government and military software providers.

However, saying that "maturity models" like CMM gained ground against or opposing Agile testing may not be right. Agile movement is a 'way of working', while CMM is a process improvement idea.

But another point of view must be considered: the operational culture of an organization. While it may be true that testers must have an ability to work in a world of uncertainty, it is also true that their flexibility must have direction.

In many cases test cultures are self-directed and as a result fruitless; unproductive results can ensue. Furthermore, providing positive evidence of defects may either indicate that you have found the tip of a much larger problem, or that you have exhausted all possibilities. A framework is a test of Testing. It provides a boundary that can measure (validate) the capacity of our work. Both sides have, and will continue to argue the virtues of their work. The proof however is in each and every assessment of delivery quality. It does little good to test systematically if you are too narrowly focused. On the other hand, finding a bunch of errors is not an indicator that Agile methods was the driving force; you may simply have stumbled upon an obviously poor piece of work.

Exploratory vs. scripted

Exploratory testing means simultaneous test design and test execution with an emphasis on learning. Scripted testing means that learning and test design happen prior to test execution, and quite often the learning has to be done again during test execution. Exploratory testing is very common, but in most writing and training about testing it is barely mentioned and generally misunderstood. Some writers consider it a primary and essential practice. Structured exploratory testing is a compromise when the testers are familiar with the software. A vague test plan, known as a test charter, is written up, describing what functionalities need to be tested but not how, allowing the individual testers to choose the method and steps of testing.

There are two main disadvantages associated with a primarily exploratory testing approach. The first is that there is no opportunity to prevent defects, which can happen when the designing of tests in advance serves as a form of structured static testing that often reveals problems in system requirements and design. The second is that, even with test charters, demonstrating test coverage and achieving repeatability of tests using a purely exploratory testing approach is difficult. For this reason, a blended approach of scripted and exploratory testing is often used to reap the benefits while mitigating each approach's disadvantages.

Manual vs. automated

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly.^[3] Others, such as advocates of agile development, recommend automating 100% of all tests. A challenge with automation is that automated testing requires automated test oracles (an oracle is a mechanism or principle by which a problem in the software can be recognized). Such tools have value in load testing software (by signing on to an application with hundreds or thousands of instances simultaneously), or in checking for intermittent errors in software. The success of automated software testing depends on complete and comprehensive test planning. Software development strategies such as test-driven development are highly compatible with the idea of devoting a large part of an organization's testing resources to automated testing. Many large software organizations perform automated testing. Some have developed their own automated testing environments specifically for internal development, and not for resale.

Software design vs. software implementation

Ideally, software testers should not be limited only to testing software implementation, but also to testing software design. With this assumption, the role and involvement of testers will change dramatically. In such an environment, the test cycle will change too. To test software design, testers would review requirement and design specifications together with designer and programmer, potentially helping to identify bugs earlier in software development.

Who watches the watchmen?

One principle in software testing is summed up by the classical Latin question posed by Juvenal: *Quis Custodiet Ipsos Custodes* (Who watches the watchmen?), or is alternatively referred informally, as the "Heisenbug" concept (a common misconception that confuses Heisenberg's uncertainty principle with observer effect). The idea is that any form of observation is also an interaction, that the act of testing can also affect that which is being tested.

In practical terms the test engineer is testing software (and sometimes hardware or firmware) with other software (and hardware and firmware). The process can fail in ways that are not the result of defects in the target but rather result from defects in (or indeed intended features of) the testing tool.

There are metrics being developed to measure the effectiveness of testing. One method is by analyzing code coverage (this is highly controversial) - where everyone can agree what areas are not being covered at all and try to improve coverage in these areas.

Bugs can also be placed into code on purpose, and the number of bugs that have not been found can be predicted based on the percentage of intentionally placed bugs that were found. The problem is that it assumes that the intentional bugs are the same type of bug as the unintentional ones.

Finally, there is the analysis of historical find-rates. By measuring how many bugs are found and comparing them to predicted numbers (based on past experience with similar projects), certain assumptions regarding the effectiveness of testing can be made. While not an absolute measurement of quality, if a project is halfway complete and there have been no defects found, then changes may be needed to the procedures being employed by QA.

References

- [1] context-driven-testing.com (<http://www.context-driven-testing.com>)
- [2] Kaner, Cem; Jack Falk, Hung Quoc Nguyen (1993). *Testing Computer Software* (Third Edition ed.). John Wiley and Sons. ISBN 1-85032-908-7.
- [3] An example is Mark Fewster, Dorothy Graham: *Software Test Automation*. Addison Wesley, 1999, ISBN 0-201-33140-3

Software testing life cycle

Software testing life cycle identifies what test activities to carry out and when (what is the best time) to accomplish those test activities. Even though testing differs between organizations, there is a testing life cycle.

Software Testing Life Cycle consists of six (generic) phases:

- Test Planning,
- Test Analysis,
- Test Design,
- Construction and verification,
- Testing Cycles,
- Final Testing and Implementation and
- Post Implementation.

Software testing has its own life cycle that intersects with every stage of the SDLC. The basic requirements in software testing life cycle is to control/deal with software testing – Manual, Automated and Performance.

Test Planning

This is the phase where Project Manager has to decide what things need to be tested, do I have the appropriate budget etc. Naturally proper planning at this stage would greatly reduce the risk of low quality software. This planning will be an ongoing process with no end point.

Activities at this stage would include preparation of high level test plan-(according to IEEE test plan template The Software Test Plan (STP) is designed to prescribe the scope, approach, resources, and schedule of all testing activities. The plan must identify the items to be tested, the features to be tested, the types of testing to be performed, the personnel responsible for testing, the resources and schedule required to complete testing, and the risks associated with the plan.). Almost all of the activities done during this stage are included in this software test plan and revolve around a test plan.

In Test Planning following are the major tasks: 1. Defining scope of testing 2. Identification of approaches 3. Defining risk 4. Identifying resources 5. Defining Time Schedule

Test Analysis

Once test plan is made and decided upon, next step is to deal a little more into the project and decide what types of testing should be carried out at different stages of SDLC, do we need or plan to automate, if yes then when the appropriate time to automate is, what type of specific documentation I need for testing.

Proper and regular meetings should be held between testing teams, project managers, development teams, Business Analysts to check the progress of things which will give a fair idea of the movement of the project and ensure the completeness of the test plan created in the planning phase, which will further help in enhancing the right testing strategy created earlier. We will start creating test case formats and test cases itself. In this stage we need to develop Functional validation matrix based on Business Requirements to ensure that all system requirements are covered by one or more test cases, identify which test cases to automate, begin review of documentation, i.e. Functional Design, Business Requirements, Product Specifications, Product Externals etc. We also have to define areas for Stress and Performance Testing.

Test Design

Test plans and cases which were developed in the analysis phase are revised. Functional validation matrix is also revised and finalized. In this stage risk assessment criteria is developed. If you have thought of automation then you have to select which test cases to automate and begin writing scripts for them. Test data is prepared. Standards for unit testing and pass / fail criteria are defined here. Schedule for testing is revised (if necessary) & finalized and test environment is prepared.

Construction and verification

In this phase we have to complete all the test plans, test cases, complete the scripting of the automated test cases, Stress and Performance testing plans needs to be completed. We have to support the development team in their unit testing phase. And obviously bug reporting would be done as when the bugs are found. Integration tests are performed and errors (if any) are reported.

Testing Cycles

In this phase we have to complete testing cycles until test cases are executed without errors or a predefined condition is reached. Run test cases --> Report Bugs --> revise test cases (if needed) --> add new test cases (if needed) --> bug fixing --> retesting (test cycle 2, test cycle 3....).

Final Testing and Implementation

In this we have to execute remaining stress and performance test cases, documentation for testing is completed / updated, provide and complete different matrices for testing. Acceptance, load and recovery testing will also be conducted and the application needs to be verified under production conditions.

Post Implementation

In this phase, the testing process is evaluated and lessons learnt from that testing process are documented. Line of attack to prevent similar problems in future projects is identified. Create plans to improve the processes. The recording of new errors and enhancements is an ongoing process. Cleaning up of test environment is done and test machines are restored to base lines in this stage.

==Software Testing Life Cycle==h Phase Activities Outcome Planning Create high level test plan Test plan, Refined Specification Analysis

Create detailed test plan, Functional Validation Matrix, test cases
Revised Test Plan, Functional validation matrix, test cases

Design

test cases are revised; select which test cases to automate
revised test cases, test data sets, sets, risk assessment sheet

Construction

scripting of test cases to automate,
test procedures/Scripts, Drivers, test results, Bugreports.

Testing cycles complete testing cycles Test results, Bug Reports Final testing execute remaining stress and performance tests, complete documentation Test results and different metrics on test efforts Post implementation Evaluate testing processes Plan for improvement of testing process

Software testing lifecycle is a systematic approach towards the sequence of activities conducted during Testing phase. 1.Test Planning 2.Test Development 3.Test Execution 4.Result Analysis 5.Bug Tracking 6.Reporting.

Software testing outsourcing

Software testing outsourcing provides for software testing carried out by the forces of an additionally engaged company or a group of people not directly involved in the process of software development. Contemporary testing outsourcing is an independent IT field, the so called Software Testing & Quality Assurance.

Software testing is an essential phase of software development, but is definitely not the core activity of most companies. Outsourcing enables the company to concentrate on its core activities while external software testing experts handle the independent validation work. This offers many tangible business benefits. These include independent assessment leading to enhanced delivery confidence, reduced time to market, lower infrastructure investment, predictable software quality, de-risking of deadlines and increased time to focus on designing better solutions. Today stress, performance and security testing are the most demanded types in software testing outsourcing.

At present **5 main options of software testing outsourcing** are available depending on the detected problems with software development:

- full outsourcing of the whole palette of software testing & quality assurance operations
- realization of complex testing with high resource consumption
- prompt resource enlargement of the company by external testing experts
- support of existing program products by new releases testing
- independent quality audit.

Availability of the effective channels of communication and information sharing is one of the core aspects that allow to guarantee the high quality of testing, being at the same time the main obstacle for outsourcing. Due to this channels software testing outsourcing allows to cut down the number of software defects 3 – 30 times depending on the quality of the legacy system.

Top established global outsourcing cities

According to Tholons Global Services - Top 50 ^[1], in 2009, Top Established and Emerging Global Outsourcing Cities in Testing function were:

1. Cebu City, Philippines
2. Shanghai, China
3. Beijing, China
4. Kraków, Poland
5. Ho Chi Minh City, Vietnam

Top Emerging Global Outsourcing Cities

1. Bucharest
2. São Paulo
3. Cairo

Cities were benchmark against six categories included: skills and scalability, savings, business environment, operational environment, business risk and non-business environment.

Vietnam Outsourcing

Vietnam has become a major player in software outsourcing. Tholon's Global Services annual report highlights Ho Chi Minh City ability to competitively meet client nations' needs in scale and capacity. Its rapid maturing business environment has caught the eye of international investors aware of the country's stability in political and labor conditions, its increasing number of English speakers and its high service-level maturity ^[2].

Californian based companies such as Global CyberSoft Inc. and LogiGear Corporation are optimistic with Vietnam's ability to execute their global offshoring industry requirements. Despite the 2008-2009 financial crisis, both companies expect to fulfill their projected goals. LogiGear has addressed a shortage of highly qualified software technicians for its testing and automation services but remains confident that professionals are available to increase its staff in anticipation of the US recovery ^[2].

References

- [1] Tholons Global Services report 2009 (http://www.itida.gov.eg/Documents/Tholons_study.pdf) Top Established and Emerging Global Outsourcing
- [2] (<http://www.logigear.com/in-the-news/974-software-outsourcing-recovery-and-development.html>) LogiGear, PC World Viet Nam, Jan 2011

Software Testing, Verification & Reliability

Software Testing, Verification & Reliability

Editor	Jeff Offutt
Categories	Computer science
Frequency	Quarterly
Company	Wiley
Country	United States
Language	English
Website	interscience.wiley.com ^[1]
ISSN	0960-0833 ^[2]

Software Testing, Verification & Reliability (STVR) is a leading journal in the field of software testing, verification, and reliability. STVR is a quarterly international journal that included papers on both theoretical and practical issues.

For 13 years, until 2006, the Chief Editor was Martin Woodward.^[3] The current editor is Jeff Offutt.^[4]

The journal is published by John Wiley & Sons.

References

- [1] <http://www3.interscience.wiley.com/journal/13635/home>
- [2] <http://www.worldcat.org/issn/0960-0833>
- [3] A Tribute to Martin Woodward (<http://doi.wiley.com/10.1002/stvr.363>), *Software Testing, Verification & Reliability*, 16:209–211, 2006. doi:10.1002/stvr.363
- [4] Martin R. Woodward and Jeff Offutt, Editorial: Reflections on the past, present and future. *Software Testing, Verification & Reliability*, 17(1):1–2, 2007.

External links

- STVR journal homepage (<http://www3.interscience.wiley.com/journal/13635/home>)
- STVR journal information (<http://eu.wiley.com/WileyCDA/WileyTitle/productCd-STVR.html>)
- STVR information (<http://www.informatik.uni-trier.de/~ley/db/journals/stvr/>) from DBLP

Software verification

Software verification is a broader and more complex discipline of software engineering whose goal is to assure that software fully satisfies all the expected requirements.

There are two fundamental approaches to verification:

- *Dynamic verification*, also known as Test or Experimentation - This is good for finding bugs
- *Static verification*, also known as Analysis - This is useful for proving correctness of a program although it may result in false positives

Dynamic verification (Test, experimentation)

Dynamic verification is performed during the execution of software, and dynamically checks its behaviour; it is commonly known as the Test phase. Verification is a Review Process. Depending on the scope of tests, we can categorize them in three families:

- *Test in the small*: a test that checks a single function or class (Unit test)
- *Test in the large*: a test that checks a group of classes, such as
 - Module test (a single module)
 - Integration test (more than one module)
 - System test (the entire system)
- *Acceptance test*: a formal test defined to check acceptance criteria for a software
 - Functional test
 - Non functional test (performance, stress test)

Software verification is often confused with software validation. The difference between **verification and validation**:

- Software *verification* asks the question, "Are we building the product right?"; that is, does the software conform to its specification.
- Software *validation* asks the question, "Are we building the right product?"; that is, is the software doing what the user really requires.

The aim of software verification is to find the errors introduced by an activity, i.e. check if the product of the activity is as correct as it was at the beginning of the activity.

Static verification (Analysis)

Static verification is the process of checking that software meets requirements by doing a physical inspection of it. For example:


- *Code conventions verification*
 - *Bad practices (anti-pattern) detection*
 - Software metrics calculation
 - Formal verification
-

References

- IEEE: *SWEBOK: Guide to the Software Engineering Body of Knowledge*
 - Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli: *Fundamentals of Software Engineering*, Prentice Hall, ISBN 0-13-099183-X
 - Alan L. Breitler: *A Verification Procedure for Software Derived from Artificial Neural Networks*, Journal of the International Test and Evaluation Association, Jan 2004, Vol 25, No 4.
-

Sputnik (JavaScript conformance test)

Sputnik

	
URL	sputnik.googlelabs.com ^[1]
Commercial?	No
Registration	No
Content license	New BSD License
Owner	Google Inc.
Created by	Christian Plesner Hansen, Sandholm
Launched	29 June 2009 ^[2]

Sputnik is a JavaScript conformance test suite. The purpose of the test suite is to determine how well a JavaScript implementation adheres to the ECMA-262 specification, 5th edition, looking only at those features that were also present in the 3rd edition. ^[2] It contains over 5000 tests that touch all aspects of the JavaScript language. ^[2] ^[3]

The test was created in Russia for testing the conformance of the V8 JavaScript engine used in Google Chrome. ^[2]

As part of phasing out Google Labs, Google will be shutting down Sputnik by August 31, 2011. All current Sputnik tests have been incorporated into Ecma's test262 test suite.

Browsers that do not pass

There is currently no browser passing the Sputnik test. ^[3]

Desktop browsers

Scores represent the number of failed tests - a perfect score is 0 (100%).

Desktop browser results in Sputnik

Browser name	Score of current release	Score of preview release
Internet Explorer	Internet Explorer 9 72/5246 (98.63%)	Internet Explorer 10 PP2 71/5246 (98.65%)
Google Chrome	Google Chrome 12.0.742.112 136/5246 (97.41%)	Google Chrome 15.0.854 134/5246 (97.45%)
Safari	Safari 5.1 142/5246 (97.29%)	No preview available

Mozilla Firefox	Firefox 6.0 128/5246 (97.56%)	Firefox 9.0a1 (Build: 2011-08-20) 95/5246 (98.19%)
Opera	Opera 11.50 (build 1074) 74/5246 (98.59%)	Opera 12.00 (build 1033) 80/5246 (98.48%)

Ecmascript testsuite

Google has handed the test from Sputnik testsuite to Ecma International for inclusion in its EcmaScript 262 testsuite [4]. Some sputnik tests however have been found to have issues and are not fully conforming to EcmaScript version 5 specification. [5]

Mobile browsers

Mobile browsers

Browser name	Score of current release	Score of preview release
Android	128/5246 (97.56%)	no preview version
Internet Explorer Mobile	Internet Explorer Mobile 7 477/5246 (90.91%)	Internet Explorer Mobile 9 (SDK emulator) 88/5246 (98.32%)

References

- [1] <http://sputnik.googlelabs.com/>
- [2] Hansen, Christian Plesner (29 June 2009). "Launching Sputnik into Orbit" (<http://blog.chromium.org/2009/06/launching-sputnik-into-orbit.html>). *The Chromium Blog*. Google. . Retrieved 8 September 2010.
- [3] De, Pallab (11th Mar 2010). "Does Your Browser Behave? Find Out With Sputnik JavaScript Conformance Test Suite" (<http://techie-buzz.com/tech-news/sputnik-javascript-test.html>). TechieBuzz. . Retrieved 8 September 2010.
- [4] <http://test262.ecmascript.org/>
- [5] Bug 18 -several tests assume it's okay to have a FunctionDeclaration in a Statement context (https://bugs.ecmascript.org/show_bug.cgi?id=18)

External links

- Source Code (<http://code.google.com/p/sputniktests/>) on Google Code

STAR (Conference)

STAR is a series of conferences regarding software testing.

The conferences are a combination of workshops, keynotes and tracks, all about one or more subjects related to software testing:

- Test Management
- Test Techniques
- Test Automation
- Metrics
- SOA Testing
- Outsourcing
- Static Testing
- Web Testing
- Exploratory Testing
- Agile Testing
- Performance testing

In the USA, SQE organises 2 conferences: STAR East and STAR West. In Europe, Qualtech Conferences organises EuroSTAR.

References

- STAR East ^[1]
- STAR West ^[2]
- EuroSTAR ^[3]
- STAR East 2008 ^[4] on Confabb
- Star East 2007 ^[5] on AllConferences.com

References

[1] <http://www.sqe.com/StarEast/>

[2] <http://www.sqe.com/StarWest/>

[3] <http://www.qualtechconferences.com/?id=2>

[4] <http://www.confabb.com/conferences/51486-stareast-2008-software-testing-analysis-review>

[5] <http://www.allconferences.com/conferences/20060913094001/>

Stream X-Machine

The **Stream X-machine (SXM)** is a model of computation introduced by Gilbert Laycock in his 1993 PhD thesis, *The Theory and Practice of Specification Based Software Testing*.^[1] Based on Samuel Eilenberg's X-machine, an extended finite state machine for processing data of the type X ,^[2] the Stream X-Machine is a kind of X-machine for processing a memory data type Mem with associated input and output streams In^* and Out^* , that is, where $X = Out^* \times Mem \times In^*$. The transitions of a Stream X-Machine are labelled by functions of the form $\varphi: Mem \times In \rightarrow Out \times Mem$, that is, which compute an output value and update the memory, from the current memory and an input value.

Although the general X-machine had been identified in the 1980s as a potentially useful formal model for specifying software systems,^[3] it was not until the emergence of the Stream X-Machine that this idea could be fully exploited. Florentin Ipate and Mike Holcombe went on to develop a theory of *complete* functional testing,^[4] in which complex software systems with hundreds of thousands of states and millions of transitions could be decomposed into separate SXMs that could be tested exhaustively, with a guaranteed proof of correct integration.^[5]

Because of the intuitive interpretation of Stream X-Machines as "processing agents with inputs and outputs", they have attracted increasing interest, because of their utility in modelling real-world phenomena. The SXM model has important applications in fields as diverse as computational biology, software testing and agent-based computational economics.

The Stream X-Machine

A Stream X-Machine (SXM) is an extended finite state machine with auxiliary memory, inputs and outputs. It is a variant of the general X-machine, in which the fundamental data type $X = Out^* \times Mem \times In^*$, that is, a tuple consisting of an output stream, the memory and an input stream. A SXM separates the *control flow* of a system from the *processing* carried out by the system. The control is modelled by a finite state machine (known as the *associated automaton*) whose transitions are labelled with processing functions chosen from a set Φ (known as the *type* of the machine), which act upon the fundamental data type.

Each processing function in Φ is a partial function, and can be considered to have the type $\varphi: Mem \times In \rightarrow Out \times Mem$, where Mem is the memory type, and In and Out are respectively the input and output types. In any given state, a transition is *enabled* if the domain of the associated function φ_i includes the next input value and the current memory state. If (at most) one transition is enabled in a given state, the machine is *deterministic*. Crossing a transition is equivalent to applying the associated function φ_i , which consumes one input, possibly modifies the memory and produces one output. Each recognised path through the machine therefore generates a list $\varphi_1 \dots \varphi_n$ of functions, and the SXM composes these functions together to generate a relation on the fundamental data type $\{\varphi_1 \dots \varphi_n\}: X \rightarrow X$.

Relationship to X-machines

The Stream X-Machine is a variant of X-machine in which the fundamental data type $X = Out^* \times Mem \times In^*$. In the original X-machine, the φ_i are general *relations* on X . In the Stream X-Machine, these are usually restricted to *functions*; however the SXM is still only deterministic if (at most) one transition is enabled in each state.

A general X-machine handles input and output using a prior encoding function $\alpha: Y \rightarrow X$ for input, and a posterior decoding function $\beta: X \rightarrow Z$ for output, where Y and Z are respectively the input and output types. In a Stream X-Machine, these types are streams:

$$\begin{aligned} Y &= In^* \\ Z &= Out^* \end{aligned}$$

and the encoding and decoding functions are defined as:

$$\begin{aligned}\alpha(\text{ins}) &= (\langle \rangle, \text{mem}_0, \text{ins}) \\ \beta(\text{outs}, \text{mem}_n, \langle \rangle) &= \text{outs}\end{aligned}$$

where $\text{ins}: In^*$, $\text{outs}: Out^*$ and $\text{mem}_i: Mem$. In other words, the machine is initialized with the whole of the input stream; and the decoded result is the whole of the output stream, provided the input stream is eventually consumed (otherwise the result is undefined).

Each processing function in a SXM is given the abbreviated type $\varphi_{\text{SXM}}: Mem \times In \rightarrow Out \times Mem$. This can be mapped onto a general X-machine relation of the type $\varphi: X \rightarrow X$ if we treat this as computing:

$$\varphi(\text{outs}, \text{mem}_i, \text{in} :: \text{ins}) = (\text{outs} :: \text{out}, \text{mem}_{i+1}, \text{ins})$$

where $::$ denotes concatenation of an element and a sequence. In other words, the relation extracts the head of the input stream, modifies memory and appends a value to the tail of the output stream.

Processing and Testable Properties

Because of the above equivalence, attention may focus on the way a Stream X-Machine processes inputs into outputs, using an auxiliary memory. Given an initial memory state mem_0 and an input stream ins , the machine executes in a step-wise fashion, consuming one input at a time, and generating one output at a time. Provided that (at least) one recognised path $\text{path} = \varphi_1 \dots \varphi_n$ exists leading to a state in which the input has been consumed, the machine yields a final memory state mem_n and an output stream outs . In general, we can think of this as the relation computed by all recognised paths: $|\text{path}|: In^* \rightarrow Out^*$. This is often called the *behaviour* of the Stream X-Machine.

The behaviour is deterministic, if (at most) one transition is enabled in each state. This property, and the ability to control how the machine behaves in a step-wise fashion in response to inputs and memory, makes it an ideal model for the specification of software systems. If the specification and implementation are both assumed to be Stream X-Machines, then the implementation may be tested for conformance to the specification machine, by observing the inputs and outputs at each step. Laycock first highlighted the utility of single-step processing with observations for testing purposes.^[1]

Holcombe and Ipaté developed this into a practical theory of software testing^[4] which was fully compositional, scaling up to very large systems.^[6] A proof of correct integration^[5] guarantees that testing each component and each integration layer separately corresponds to testing the whole system. This divide-and-conquer approach makes *exhaustive* testing feasible for large systems.

The testing method is described in a separate article on the Stream X-Machine testing methodology.

Applications

Stream X-Machines have been used in a number of different application areas.

External links

- The MOTIVE project^[7], using SXM techniques to generate test sets for object-oriented software.
- The EURACE project^[8], an application of CSXM techniques to agent-based computational economics.
- x-machines.net^[9], a site describing the background to X-machine research.
- Mike (Prof. W.M.L.) Holcombe^[10]'s web page at Sheffield University.

References

[1] Gilbert Laycock (1993) *The Theory and Practice of Specification Based Software Testing*. PhD Thesis, University of Sheffield, Dept of Computer Science.

Abstract (<http://www.mcs.le.ac.uk/people/gt11/PhDabstract.html>)

[2] Samuel Eilenberg (1974) *Automata, Languages and Machines, Vol. A*.

London: Academic Press.

[3] M. Holcombe (1988) 'X-machines as a basis for dynamic system specification'. *Software Engineering Journal* 3 (2), pp. 69-76.

[4] Mike Holcombe and Florentin Ipate (1998) *Correct systems - building a business process solution*. Applied Computing Series.

Berlin: Springer-Verlag.

[5] F. Ipate and W. M. L. Holcombe (1997) 'An integration testing method which is proved to find all faults'. *Int. J. Comp. Math.*, 63, pp. 159-178.

[6] F. Ipate and M. Holcombe (1998) 'A method for refining and testing generalised machine specifications'. *Int. J. Comp. Math.* 68, pp. 197-219.

[7] <http://www.dcs.shef.ac.uk/~ajhs/motive/>

[8] <http://www.dcs.shef.ac.uk/~stc/eurace/>

[9] <http://www.x-machines.net>

[10] <http://www.dcs.shef.ac.uk/~wmlh/>

Stress testing

Stress testing is a form of testing that is used to determine the stability of a given system or entity. It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results. Stress testing may have a more specific meaning in certain industries, such as fatigue testing for materials.

Computer software

In software testing, a system stress test refers to tests that put a greater emphasis on robustness, availability, and error handling under a heavy load, rather than on what would be considered correct behavior under normal circumstances. In particular, the goals of such tests may be to ensure the software does not crash in conditions of insufficient computational resources (such as memory or disk space), unusually high concurrency, or denial of service attacks.

Examples:

- A web server may be stress tested using scripts, bots, and various denial of service tools to observe the performance of a web site during peak loads.

Stress testing may be contrasted with load testing:

- Load testing examines the entire environment and database, while measuring the response time, whereas stress testing focuses on identified transactions, pushing to a level so as to break transactions or systems.
 - During stress testing, if transactions are selectively stressed, the database may not experience much load, but the transactions are heavily stressed. On the other hand, during load testing the database experiences a heavy load, while some transactions may not be stressed.
 - System stress testing, also known as stress testing, is loading the concurrent users over and beyond the level that the system can handle, so it breaks at the weakest link within the entire system.
-

Hardware

Reliability engineers often test items under expected stress or even under accelerated stress. The goal is to determine the operating life of the item or to determine modes of failure.^[1]

Stress testing, in general, should put the hardware under exaggerated levels of stress in order to ensure stability when used in a normal environment.

Computer processors

When modifying the operating parameters of a CPU, such as in overclocking, underclocking, overvolting, and undervolting, it may be necessary to verify if the new parameters (usually CPU core voltage and frequency) are suitable for heavy CPU loads. This is done by running a CPU-intensive program (usually Prime95) for extended periods of time, to test whether the computer hangs or crashes. CPU stress testing is also referred to as **torture testing**. Software that is suitable for torture testing should typically run instructions that utilise the entire chip rather than only a few of its units.

Stress testing a CPU over the course of 24 hours at 100% load is, in most cases, sufficient enough to determine that the CPU will function correctly in normal usage scenarios, where CPU usage fluctuates at low levels (50% and under), such as on a desktop computer.

Financial sector

Instead of doing financial projection on a "best estimate" basis, a company may do stress testing where they look at how robust a financial instrument is in certain crashes, a form of scenario analysis. They may test the instrument under, for example, the following stresses:

- What happens if equity markets crash by more than x% this year?
- What happens if interest rates go up by at least y%?
- What if half the instruments in the portfolio terminate their contracts in the fifth year?
- What happens if oil prices rise by 200%?

This type of analysis has become increasingly widespread, and has been taken up by various governmental bodies (such as the FSA in the UK) as a regulatory requirement on certain financial institutions to ensure adequate capital allocation levels to cover potential losses incurred during extreme, but plausible, events. This emphasis on adequate, risk adjusted determination of capital has been further enhanced by modifications to banking regulations such as Basel II. Stress testing models typically allow not only the testing of individual stressors, but also combinations of different events. There is also usually the ability to test the current exposure to a known historical scenario (such as the Russian debt default in 1998 or 9/11 attacks) to ensure the liquidity of the institution.

Stress testing reveals how well a portfolio is positioned in the event forecasts prove true. Stress testing also lends insight into a portfolio's vulnerabilities. Though extreme events are never certain, studying their performance implications strengthens understanding.

Defining stress tests

Stress testing defines a scenario and uses a specific algorithm to determine the expected impact on a portfolio's return should such a scenario occur. There are three types of scenarios:

- Extreme event: hypothesize the portfolio's return given the recurrence of a historical event. Current positions and risk exposures are combined with the historical factor returns.
- Risk factor shock: shock any factor in the chosen risk model by a user-specified amount. The factor exposures remain unchanged, while the covariance matrix is used to adjust the factor returns based on their correlation with the shocked factor.

- External factor shock: instead of a risk factor, shock any index, macro-economic series (e.g., oil prices), or custom series (e.g., exchange rates). Using regression analysis, new factor returns are estimated as a result of the shock.

In an exponentially weighted stress test, historical periods more like the defined scenario receive a more significant weighting in the predicted outcome. The defined decay rate lets the tester manipulate the relative importance of the most similar periods. In the standard stress test, each period is equally weighted.

Stress tests in payment and settlement systems

An other form of financial stress testing is the stress testing of financial infrastructure. As part of Central Banks' market infrastructure oversight functions, stress tests have been applied to payment and securities settlement systems.^{[2] [3] [4]} Since ultimately, the Banks need to meet their obligations in Central Bank money held in payment systems that are commonly operated or closely supervised by central banks^[5] (e.g. CHAPS, FedWire, Target2, which are also referred to as large value payment systems), it is of great interest to monitor these systems' participants' (mainly banks) liquidity positions.

The amount of liquidity held by banks on their accounts can be a lot less (and usually is) than the total value of transferred payments during a day. The total amount of liquidity needed by banks to settle a given set of payments is dependent on the balancedness of the circulation of money from account to account (reciprocity of payments), the timing of payments and the netting procedures used.^[6] The inability of some participants to send payments can cause severe falls in settlement ratios of payments. The failure of one participant to send payments can have negative contagion effects on other participants' liquidity positions and their potential to send payments.

By using stress tests it is possible to evaluate the short term effects of events such as bank failures or technical communication breakdowns that lead to the inability of chosen participants to send payments. These effects can be viewed as direct effects on the participant, but also as systemic contagion effects. How hard the other participants will be hit by a chosen failure scenario will be dependent on the available collateral and initial liquidity of participants, and their potential to bring in more liquidity.^[7] Stress test conducted on payment systems help to evaluate the short term adequacy and sufficiency of the prevailing liquidity levels and buffers of banks, and the contingency measures of the studied payment systems.^[8]

Nuclear power plants

After the 2011 earthquake in Japan the European Commission decided that all nuclear power plants in Europe have to undergo a stress test to verify they still comply with the highest safety standards.

After WENRA released the first proposal of a stress test, there has been criticism that the stress test was not going to be strict enough^[9].

References

- [1] Nelson, Wayne B., (2004), *Accelerated Testing - Statistical Models, Test Plans, and Data Analysis*, John Wiley & Sons, New York, ISBN 0-471-69736-2
- [2] Humphrey, D. (1986). Payments finality and the risks of settlement failure. In A. Saunders and L.J. White (Ed.), *Technology and the Regulation of Financial Markets: Securities, Futures and Banking* (pp. 97-120). Heath, Lexington. MA.
- [3] H. Leinonen (ed.): Simulation analyses and stress testing of payment networks (Bank of Finland Studies E:42/2009) Simulation publications (<http://pss.bof.fi/Pages/Publications.aspx>)
- [4] H. Leinonen (ed.): Liquidity, risks and speed in payment and settlement systems - a simulation approach (Bank of Finland Studies E:31/2005) Simulation publications (<http://pss.bof.fi/Pages/Publications.aspx>)
- [5] CPSS Publications No 34 (2001): CORE PRINCIPLES FOR SYSTEMICALLY IMPORTANT PAYMENT SYSTEMS (<http://www.bis.org/publ/cpss34e.htm>)
- [6] CPSS Publications (1990): Report of the Committee on Interbank Netting Schemes of the Central Banks of the Group of Ten countries (<http://www.bis.org/publ/cpss04.htm>)
- [7] ECB: EU Banks' Liquidity Stress testing and contingency funding plans 2008 (<http://www.ecb.int/pub/pdf/other/eubankliquiditystress testing200811en.pdf>)

- [8] ECB: EU Banks' Liquidity Stress testing and contingency funding plans 2008 (<http://www.ecb.int/pub/pdf/other/eubankliquiditystress-testing200811en.pdf>)
- [9] <http://www.europeanvoice.com/article/2011/may/eu-to-decide-nuclear-stress-test-criteria-next-week/70962.aspx>

External links

- The Bank of Finland Payment and Settlement Simulator BoF-PSS (<http://www.bof.fi/sc/bof-pss/>)

Stress testing (software)

In software testing, **stress testing** refers to tests that determine the robustness of software by testing beyond the limits of normal operation. Stress testing is particularly important for "mission critical" software, but is used for all types of software. Stress tests commonly put a greater emphasis on robustness, availability, and error handling under a heavy load, than on what would be considered correct behavior under normal circumstances.

Field experience

Failures may be related to:

- use of non production like environments, e.g. databases of smaller size
- complete lack of load or stress testing

Rationale

Reasons for stress testing include:

- The software being tested is "mission critical", that is, failure of the software (such as a crash) would have disastrous consequences.
- The amount of time and resources dedicated to testing is usually not sufficient, with traditional testing methods, to test all of the situations in which the software will be used when it is released.
- Even with sufficient time and resources for writing tests, it may not be possible to determine beforehand all of the different ways in which the software will be used. This is particularly true for operating systems and middleware, which will eventually be used by software that doesn't even exist at the time of the testing.
- Customers may use the software on computers that have significantly fewer computational resources (such as memory or disk space) than the computers used for testing.
- Concurrency is particularly difficult to test with traditional testing methods. Stress testing may be necessary to find race conditions and deadlocks.
- Software such as web servers that will be accessible over the Internet may be subject to denial of service attacks.
- Under normal conditions, certain types of bugs, such as memory leaks, can be fairly benign and difficult to detect over the short periods of time in which testing is performed. However, these bugs can still be potentially serious. In a sense, stress testing for a relatively short period of time can be seen as simulating normal operation for a longer period of time.

Relationship to branch coverage

Branch coverage (a specific type of code coverage) is a metric of the number of branches executed under test, where "100% branch coverage" means that every branch in a program has been executed at least once under some test. Branch coverage is one of the most important metrics for software testing; software for which the branch coverage is low is not generally considered to be thoroughly tested. Note that code coverage metrics are a property of the tests for a piece of software, not of the software being tested.

Achieving high branch coverage often involves writing *negative* test variations, that is, variations where the software is supposed to fail in some way, in addition to the usual *positive* test variations, which test intended usage. An example of a negative variation would be calling a function with illegal parameters. There is a limit to the branch coverage that can be achieved even with negative variations, however, as some branches may only be used for handling of errors that are beyond the control of the test. For example, a test would normally have no control over memory allocation, so branches that handle an "out of memory" error are difficult to test.

Stress testing can achieve higher branch coverage by producing the conditions under which certain error handling branches are followed. The coverage can be further improved by using fault injection.

Examples

- A web server may be stress tested using scripts, bots, and various denial of service tools to observe the performance of a web site during peak loads.

System integration testing

Definition

System integration testing is the process of verifying the synchronization between two or more software systems and which can be performed after software system collaboration is completed....

Introduction

It is part of the software testing life cycle for software collaboration involving projects. Such software is where consumers run system integration test (SIT) round before the user acceptance test (UAT) round. And software providers usually run a pre-SIT round before Software consumers run their SIT test cases.

As an example if we are providing a solution for a software consumer as enhancement to their existing solution, then we should integrate our application layer and our DB layer with consumer's existing application and existing DB layers. After the integration process completed both software systems should be synchronized.

Which means when end users use software provider's part of the integrated application (extended part) then software provider's data layer might be updated than consumer's system. And when end users use consumer's part of the integrated application (existing part) then consumer's data layer might be updated than software provider's system. Then there should be a process to exchange data imports and exports between two parties. This data exchange process should keep both systems up-to-date.

Purpose of the System integration testing is to make sure whether these systems are successfully integrated and been up-to-date by exchanging data with each other.

Overview

Integration layer keeps synchronization between two parties is a simple system integration arrangement. Usually there are software consumers and their customer parties (third party organizations) come in to action. Then software providers should keep synchronization among software provider, software consumer party and software consumer's customer parties. Software providers and software consumers should run test cases to verify the synchronization among all the systems after software system collaboration completed.

System Integration Testing (SIT), in the context of software systems and software engineering, is a testing process that exercises a software system's coexistence with others. System integration testing takes multiple integrated systems that have passed system testing as input and tests their required interactions. Following this process, the

deliverable systems are passed on to acceptance testing..... sadadadad

System Integration testing - Data driven method

This is a simple method which can perform with minimum usage of the software testing tools. Exchange some data imports and data exports. And then investigate the behavior of each data field within each individual layer. There are three main states of data flow after the software collaboration has done.

- **Data state within the integration layer**

Integration layer can be a middleware or web service(s) which is act as a media for data imports and data exports. Perform some data imports and exports and check following steps.

1. Cross check the data properties within the Integration layer with technical/business specification documents.

- If web service involved with the integration layer then we can use WSDL and XSD against our web service request for the cross check.

- If middleware involved with the integration layer then we can use data mappings against middleware logs for the cross check.

2. Execute some unit tests. Cross check the data mappings (data positions, declarations) and requests (character length, data types) with technical specifications.

3. Investigate the server logs/middleware logs for troubleshooting.

(Reading knowledge of WSDL, XSD, DTD, XML, and EDI might be required for this)

- **Data state within the database layer**

1. First check whether all the data have committed to the database layer from the integration layer.

2. Then check the data properties with the table and column properties with relevant to technical/business specification documents.

3. Check the data validations/constrains with business specification documents.

4. If there are any processing data within the database layer then check Stored Procedures with relevant specifications.

5. Investigate the server logs for troubleshooting.

(Knowledge in SQL and reading knowledge in Stored Procedures might be required for this)

- **Data state within the Application layer**

There is not that much to do with the application layer when we perform a system integration testing.

1. Mark all the fields from business requirement documents which should be visible in the UI.

2. Create a data map from database fields to application fields and check whether necessary fields are visible in UI.

3. Check data properties by some positive and negative test cases.

There are many combinations of data imports and export which we can perform by considering the time period for system integration testing

(We have to select best combinations to perform with the limited time). And also we have to repeat some of the above steps in order to test those combinations.

System testing

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic. ^[1]

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called *assemblages*) or between any of the *assemblages* and the hardware. System testing is a more limited type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

Testing the whole system

System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing tests not only the design, but also the behaviour and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).

Types of tests to include in system testing

The following examples are different types of testing that should be considered during System testing:

- Graphical user interface testing
- Usability testing
- Performance testing
- Compatibility testing
- Error handling testing
- Load testing
- Volume testing
- Stress testing
- Security testing
- Scalability testing
- Sanity testing
- Smoke testing
- Exploratory testing
- Ad hoc testing
- Regression testing
- Reliability testing
- Installation testing
- Maintenance testing
- Recovery testing and failover testing.
- Accessibility testing, including compliance with:
 - Americans with Disabilities Act of 1990
 - Section 508 Amendment to the Rehabilitation Act of 1973
 - Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C)

Although different testing organizations may prescribe different tests as part of System testing, this list serves as a general framework or foundation to begin with.

References

[1] *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*; IEEE; New York, NY.; 1990.

- Black, Rex; (2002). *Managing the Testing Process* (2nd ed.). Wiley Publishing. ISBN 0-471-22398-0

Tessy (Software)

Tessy is a tool, that automates module/unit testing of embedded software written in various embedded dialects of the C/C++ programming language. Tessy can be used to get the certification for product according to standards like IEC 61508, EN 50128/50129, DO-178B, Automotive SPiCE, or the General Principles of Software Validation by the FDA.

Tessy is developed by Razorcat Development. Hitex Development Tools is the international sales partner.

Key Features

Automated test execution

Test report generation

Code coverage without extra effort

Regression and integration testing

Essential to get certifications

For geographically distributed projects

Testing on host or actual hardware

Supports C and C++

Characteristics

Tessy automatically executes tests, evaluates the test results, and generates the test reports. This makes Tessy suitable for regression testing. Tessy includes the Classification Tree Editor CTE, a tool to support the Classification Tree Method for Test Case Specification. Tessy supports test documentation in various formats, e.g. HTML, Word, Excel.

Origin

Both Tessy and the CTE originate from the former software technology laboratory of Daimler-Benz in Berlin, Germany.

External Links

- Tessy at www.hitex.com [1]
- White Paper Unit Test of Embedded Software (PDF) [2]
- Hitex Development Tools [3]
- Razorcat Development [4]

References

[1] <http://www.hitex.com/perm/tessy.html>

[2] <http://www.hitex.com/fileadmin/pdf/products/tessy/white-papers/WP-TESSY-0101.pdf>

[3] <http://www.hitex.com>

[4] <http://www.razorcat.com>

Test Anything Protocol

The **Test Anything Protocol** (TAP) is a protocol to allow communication between unit tests and a test harness. It allows individual tests (TAP producers) to communicate test results to the testing harness in a language-agnostic way. Originally developed for unit testing of the Perl interpreter in 1987, producers and parsers are now available for many development platforms.

History

TAP was created for the first version of Perl (released in 1987), as part of the Perl's core test harness (`t/TEST`). The `Test::Harness` module was written by Tim Bunce and Andreas König to allow Perl module authors to take advantage of TAP.

Development of TAP, including standardization of the protocol, writing of test producers and consumers, and evangelizing the language is coordinated at the TestAnything website^[1].

Specification

Despite being about 20 years old and widely used, no formal specification exists for this protocol. The behavior of the `Test::Harness` module is the de-facto TAP standard, along with a writeup of the specification on CPAN^[2].

A project to produce an IETF standard for TAP was initiated in August 2008, at YAPC::Europe 2008.^[1]

Usage examples

TAP's general format is:

```
1..N
ok 1 Description # Directive
# Diagnostic
....
ok 47 Description
ok 48 Description
more tests....
```

For example, a test file's output might look like:

```
1..4
ok 1 - Input file opened
not ok 2 - First line of the input valid.
    More output from test 2. There can be
    arbitrary number of lines for any output
    so long as there is at least some kind
    of whitespace at beginning of line.
ok 3 - Read the rest of the file
#TAP meta information
not ok 4 - Summarized correctly # TODO Not written yet
```

External links

- <http://testanything.org/>^[3] is a site dedicated to the discussion, development and promotion of TAP.

List of TAP Parsers

These are libraries which parse TAP and display the results.

- Test::Harness^[4] is the oldest and most complete TAP parser. It is limited in how it displays TAP. Though it most often runs tests written in Perl, it can launch any process which generates TAP. Most of the TAP spec is taken from the behavior of Test::Harness.
 - The original Test::Harness has now been deprecated, the new Test::Harness provides a minimal compatibility layer with previous behavior, but any new development shouldn't use this module, rather the TAP::Harness module.
- The t/TEST parser contained in the Perl source code.
- Test::Harness^[4] is a new and more flexible parser being written by Curtis "Ovid" Poe, Andy Armstrong and other people. It is a wrapper around TAP::Parser^[5].
- Test::Run^[6] is a fork of Test::Harness being written by Shlomi Fish.
- test-harness.php^[7] A TAP parser for PHP.
- nqpTAP^[8] A TAP parser written in NotQuitePerl (NQP), a smaller subset of the Perl 6 language.
- Tapir^[9] A TAP parser written in Parrot Intermediate Representation (PIR).
- tap4j^[10] A TAP implementation for Java.

List of TAP Producers

These are libraries for writing tests which output TAP.

- Test::More^[11] is the most popular testing module for Perl 5.
- Test::Most^[12] puts the most commonly used Perl 5 testing modules needed in one place. It is a superset of Test::More.
- PHPUnit^[13] is the xUnit implementation for PHP.
- test-more.php^[14] is a testing module for PHP based on Test::More.
- test-more-php^[15] implements Test::Simple & Test::More for PHP.
- libtap^[16] is a TAP producer written in C.
- libtap++^[17] is a TAP producer for C++
- Test.Simple^[18] is a port of the Perl Test::Simple and Test::More modules to JavaScript by David Wheeler.
- PyTAP^[19] A beginning TAP implementation for Python.
- MyTAP^[20] MySQL unit test library used for writing TAP producers in C or C++
- Bacon^[21] A Ruby library that supports a spec-based syntax and that can produce TAP output
- PLUTO^[22] PL/SQL Unit Testing for Oracle
- pgTAP^[23] PostgreSQL stored procedures that emit TAP
- SnapTest^[24] A PHP unit testing framework with TAP v13 compliant output.
- etap^[25] is a simple erlang testing library that provides TAP compliant output.
- lua-TestMore^[26] is a port of the Perl Test::More framework to Lua.
- tap4j^[10] A TAP implementation for Java.
- lime^[27] A testing framework bundled with the Symfony PHP framework.
- yuittest^[28] A JavaScript testing library (standalone)

References

- [1] "The Test Anything Protocol website" (<http://www.testanything.org/>). . Retrieved 2008-09-04.
- [2] "TAP specification" (<http://search.cpan.org/~petdance/Test-Harness-2.64/lib/Test/Harness/TAP.pod>). CPAN. . Retrieved 2010-12-31.
- [3] <http://testanything.org/>
- [4] <http://search.cpan.org/dist/Test-Harness/>
- [5] <http://search.cpan.org/dist/TAP-Parser/>
- [6] <http://web-cpan.berlios.de/modules/Test-Run/>
- [7] <http://www.digitalsandwich.com/archives/52-TAP-Compliant-PHP-Testing-Harness.html>
- [8] <http://github.com/leto/nqptap>
- [9] <http://github.com/leto/tapir>
- [10] <http://www.tap4j.org/>
- [11] <http://search.cpan.org/perldoc?Test::More>
- [12] <http://search.cpan.org/dist/Test-Most/>
- [13] <http://www.phpunit.de/>
- [14] <http://shiflett.org/code/test-more.php>
- [15] <http://code.google.com/p/test-more-php/>
- [16] <http://jc.ngo.org.uk/trac-bin/trac.cgi/wiki/LibTap>
- [17] <http://github.com/Leont/libperl--/blob/master/tap++/doc/libtap%2B%2B.pod#NAME>
- [18] <http://openjsan.org/doc/t/th/theory/Test/Simple/>
- [19] <http://git.codesimply.com/?p=PyTAP.git;a=summary>
- [20] <http://www.kindahl.net/mytap/doc/>
- [21] <http://rubyforge.org/projects/test-spec>
- [22] <http://code.google.com/p/pluto-test-framework/>
- [23] <http://pgtap.projects.postgresql.org/>
- [24] <http://www.snaptest.net>
- [25] <http://github.com/ngerakines/etap/tree/master>
- [26] <http://fperrad.github.com/lua-TestMore/>
- [27] http://www.symfony-project.org/book/1_2/15-Unit-and-Functional-Testing#The%20Lime%20Testing%20Framework
- [28] <http://yuilibary.com/yuitest/>

Test automation

Compare with Manual testing.

Test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions.^[1] Commonly, test automation involves automating a manual process already in place that uses a formalized testing process.

Overview

Although manual tests may find many defects in a software application, it is a laborious and time consuming process. In addition, it may not be effective in finding certain classes of defects. Test automation is a process of writing a computer program to do testing that would otherwise need to be done manually. Once tests have been automated, they can be run quickly and repeatedly. This is often the most cost effective method for software products that have a long maintenance life, because even minor patches over the lifetime of the application can cause features to break which were working at an earlier point in time.

There are two general approaches to test automation:

- **Code-driven testing.** The public (usually) interfaces to classes, modules, or libraries are tested with a variety of input arguments to validate that the results that are returned are correct.
- **Graphical user interface testing.** A testing framework generates user interface events such as keystrokes and mouse clicks, and observes the changes that result in the user interface, to validate that the observable behavior of the program is correct.

Test automation tools can be expensive, and it is usually employed in combination with manual testing. It can be made cost-effective in the longer term, especially when used repeatedly in regression testing.

One way to generate test cases automatically is model-based testing through use of a model of the system for test case generation but research continues into a variety of alternative methodologies for doing so.

What to automate, when to automate, or even whether one really needs automation are crucial decisions which the testing (or development) team must make. Selecting the correct features of the product for automation largely determines the success of the automation. Automating unstable features or features that are undergoing changes should be avoided.^[2]

Code-driven testing

A growing trend in software development is the use of testing frameworks such as the xUnit frameworks (for example, JUnit and NUnit) that allow the execution of unit tests to determine whether various sections of the code are acting as expected under various circumstances. Test cases describe tests that need to be run on the program to verify that the program runs as expected.

Code driven test automation is a key feature of Agile software development, where it is known as Test-driven development (TDD). Unit tests are written to define the functionality *before* the code is written. Only when all tests pass is the code considered complete. Proponents argue that it produces software that is both more reliable and less costly than code that is tested by manual exploration. It is considered more reliable because the code coverage is better, and because it is run constantly during development rather than once at the end of a waterfall development cycle. The developer discovers defects immediately upon making a change, when it is least expensive to fix. Finally, code refactoring is safer; transforming the code into a simpler form with less code duplication, but equivalent behavior, is much less likely to introduce new defects.

Graphical User Interface (GUI) testing

Many test automation tools provide record and playback features that allow users to interactively record user actions and replay them back any number of times, comparing actual results to those expected. The advantage of this approach is that it requires little or no software development. This approach can be applied to any application that has a graphical user interface. However, reliance on these features poses major reliability and maintainability problems. Relabelling a button or moving it to another part of the window may require the test to be re-recorded. Record and playback also often adds irrelevant activities or incorrectly records some activities.

A variation on this type of tool is for testing of web sites. Here, the "interface" is the web page. This type of tool also requires little or no software development. However, such a framework utilizes entirely different techniques because it is reading HTML instead of observing window events.

Another variation is scriptless test automation that does not use record and playback, but instead builds a model of the application under test and then enables the tester to create test cases by simply editing in test parameters and conditions. This requires no scripting skills, but has all the power and flexibility of a scripted approach. Test-case maintenance is easy, as there is no code to maintain and as the application under test changes the software objects can simply be re-learned or added. It can be applied to any GUI-based software application.

What to test

Testing tools can help automate tasks such as product installation, test data creation, GUI interaction, problem detection (consider parsing or polling agents equipped with oracles), defect logging, etc., without necessarily automating tests in an end-to-end fashion.

One must keep satisfying popular requirements when thinking of test automation:

- Platform and OS independence
 - Data driven capability (Input Data, Output Data, Metadata)
 - Customizable Reporting (DB Access, crystal reports)
 - Easy debugging and logging
 - Version control friendly – minimal binary files
 - Extensible & Customizable (Open APIs to be able to integrate with other tools)
 - Common Driver (For example, in the Java development ecosystem, that means Ant or Maven and the popular IDEs). This enables tests to integrate with the developers' workflows.
 - Support unattended test runs for integration with build processes and batch runs. Continuous Integration servers require this.
 - Email Notifications (automated notification on failure or threshold levels). This may be the test runner or tooling that executes it.
 - Support distributed execution environment (distributed test bed)
 - Distributed application support (distributed SUT)
-

Framework approach in automation

A framework is an integrated system that sets the rules of Automation of a specific product. This system integrates the function libraries, test data sources, object details and various reusable modules. These components act as small building blocks which need to be assembled to represent a business process. The framework provides the basis of test automation and simplifies the automation effort.

Defining boundaries between automation framework and a testing tool

Tools are specifically designed to target some particular test environment. Such as: Windows automation tool, web automation tool etc. It serves as driving agent for an automation process. However, automation framework is not a tool to perform some specific task, but is an infrastructure that provides the solution where different tools can plug itself and do their job in a unified manner. Hence providing a common platform to the automation engineer doing their job.

There are various types of frameworks. They are categorized on the basis of the automation component they leverage. These are:

1. Data-driven testing
2. Modularity-driven testing
3. Keyword-driven testing
4. Hybrid testing
5. Model-based testing

Notable test automation tools

Tool name	Produced by	Latest version
TestDrive	Original Software	7.0
HP QuickTest Professional	HP	11.0
IBM Rational Functional Tester	IBM Rational	8.2.0.2
Parasoft SOAtest	Parasoft	9.0
QF-Test	Quality First Software GmbH	3.4.1
Ranorex	Ranorex GmbH	3.0
Rational robot	IBM Rational	2003
Selenium	Open source	1.0.10
HTTP Test Tool	Open source	2.0.8
SilkTest	Micro Focus	2010 R2 WS2
TestArchitect	LogiGear	6.0
TestComplete	SmartBear Software	8.5
Testing Anywhere	Automation Anywhere	7.0
TestPartner	Micro Focus	6.3
TOSCA Testsuite	TRICENTIS Technology & Consulting	7.3.0 ^[3]
Visual Studio Test Professional	Microsoft	2010
WATIR	Open source	1.6.5
WebUI Test Studio	Telerik, Inc.	2011.1

References

- [1] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. p. 74. ISBN 0470042125. .
- [2] Brian Marick. "When Should a Test Be Automated?" (<http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=ART&ObjectId=2010>). StickyMinds.com. . Retrieved 2009-08-20.
- [3] <http://tosca-testsuite.com/Newsletter/Apr11/En/Newsletter.html>
- Elfriede Dustin, et al.: *Automated Software Testing*. Addison Wesley, 1999, ISBN 0-20143-287-0
 - Elfriede Dustin, et al.: *Implementing Automated Software Testing*. Addison Wesley, ISBN 978-0321580511
 - Mark Fewster & Dorothy Graham (1999). *Software Test Automation*. ACM Press/Addison-Wesley. ISBN 978-0201331400.
 - Roman Savenkov: *How to Become a Software Tester*. Roman Savenkov Consulting, 2008, ISBN 978-0-615-23372-7
 - Hong Zhu et al. (2008). *AST '08: Proceedings of the 3rd International Workshop on Automation of Software Test* (<http://portal.acm.org/citation.cfm?id=1370042#>). ACM Press. ISBN 978-1-60558-030-2.

External links

- Automation Myths (http://www.benchmarkqa.com/pdf/papers_automation_myths.pdf) by M. N. Alam
- Generating Test Cases Automatically (http://www.osc-es.de/media/pdf/dSPACENEWS2007-3_TargetLink_EmbeddedTester_en_701.pdf)
- Practical Experience in Automated Testing (<http://www.methodsandtools.com/archive/archive.php?id=33>)
- Test Automation: Delivering Business Value (http://www.applabs.com/internal/app_whitepaper_test_automation_delivering_business_value_1v00.pdf)
- Test Automation Snake Oil (http://www.satisfice.com/articles/test_automation_snake_oil.pdf) by James Bach
- When Should a Test Be Automated? (http://www.stickyminds.com/r.asp?F=DART_2010) by Brian Marick
- Why Automation Projects Fail (http://martproservice.com/Why_Software_Projects_Fail.pdf) by Art Beall
- Guidelines for Test Automation framework (http://info.allianceglobalservices.com/Portals/30827/docs/test_automation_framework_and_guidelines.pdf)
- Advanced Test Automation (<http://www.testars.com/docs/5GTA.pdf>)
- Seven Steps to Test Automation Success (http://www.io.com/~wazmo/papers/seven_steps.html)

Test automation framework

A **test automation framework** is a set of assumptions, concepts and tools that provide support for automated software testing. The main advantage of such a framework is the low cost for maintenance. If there is change to any test case then only the test case file needs to be updated and the Driver Script and Startup script will remain the same. Ideally, there is no need to update the scripts in case of changes to the application.

Choosing the right framework/scripting technique helps in maintaining lower costs. The costs associated with test scripting are due to development and maintenance efforts. The approach of scripting used during test automation has effect on costs.

Various framework/scripting techniques are generally used:

1. Linear (procedural code, possibly generated by tools like those that use record and playback)
2. Structured (uses control structures - typically 'if-else', 'switch', 'for', 'while' conditions/ statements)
3. Data-driven (data is persisted outside of tests in a database, spreadsheet, or other mechanism)
4. Keyword-driven
5. Hybrid (two or more of the patterns above are used)

The Testing framework is responsible for:^[1]

1. defining the format in which to express expectations
2. creating a mechanism to hook into or drive the application under test
3. executing the tests
4. reporting results

Another view Automation Framework is not a tool to perform some specific task, but is an infrastructure that provides a complete solution where different tools work together in an unified manner hence providing a common platform to the automation engineer using them.

Ref: <http://code.google.com/p/vauto/>

References

- [1] "Selenium Meet-Up 4/20/2010 Elisabeth Hendrickson on Robot Framework 1of2" (<http://www.youtube.com/watch?v=qf2i-xQ3LoY>). . Retrieved 2010-09-26.

Test automation management tools

Test automation management tools are specific tools providing test automation collaborative environment that is intended to make test automation efficient, clear for stakeholders and traceable. Since test automation is becoming cross-discipline (i.e. mixes both testing and development practices), the need of specific and dedicated environment for test automation is becoming vital.

Motivation

Test automation usually lacks of reporting, analysis and providing meaningful information about project status from automation perspective. Test management systems from the other hand mostly targeted on manual effort and cannot give all the required information. Test automation management system leverages automation effort towards efficient and continuous process of delivering test execution and new working tests by:

- Making transparent, meaningful and traceable reporting for all project stakeholders
- Easing test debugging through built-in test results analysis workflow
- Providing valuable metrics and KPIs – both technical and business-wise (trend analysis, benchmarking, gap analysis, root cause analysis, risk point analysis)
- Grid benchmarking and comparison of test execution days reduces analysis and review effort
- Clean traceability with other testing artifacts (test cases, data, issues, etc)
- Keeping historical data in a single place, easily retrievable
- Post project analysis and automation performance assessment (basically progress of test coverage shows the group performance)

Compliance with Agile

Test automation management tools are perfectly fit Agile methodologies and SDLC. In most cases test automation is to cover continuous changes in order to minimize manual regression testing, therefore at glance reporting is essential to be up to date and move project quickly. The changes are usually noted by seeing difference of errors in test logs between day A and day A+1. For example, difference in number of failures (logs errors) signal about probable changes either in AUT or in test code (broken test code base, instabilities) or rarely in both. Quick notice of changes and unified workflow of results analysis, ultimately, reduces cost of testing overall and moreover increase confidence on project quality attributes with clean reporting on hand.

TDD

Test-driven development utilizes test automation as primary driver to rapid and high-quality software production. Concepts of green line and thoughtful design supported with test before actual coding assume having special tools to track, analyze and make right decision within TDD process.

Continuous Integration

Another proper test automation practice ^[1] is being part of continuous integration which explicitly supposes to have automated test suites as final stage upon building, deployment and distributing new version of software. Basically, based on acceptance test results, a build is declared either as qualified for further testing or not qualified (rejected).^[2] CI web dashboards provide all relevant information on all stages of software building including automation test results. However, CI dashboard does not support comprehensive operations and views for automation engineer. This is another reason for having dedicated management tool which can supply high-level data to other project management tools such as CI, test management tools, issue management, change management.

References

- [1] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. ISBN 0470042125. .
- [2] Fowler, Martin. "Continuous Integration" (<http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration>). . Retrieved 2009-11-11.

Test bench

A **test bench** is a virtual environment used to verify the correctness or soundness of a design or model (e.g., a software product).

The term has its roots in the testing of electronic devices, where an engineer would sit at a lab bench with tools for measurement and manipulation, such as oscilloscopes, multimeters, soldering irons, wire cutters, and so on, and manually verify the correctness of the device under test.

In the context of software or firmware or hardware engineering, a test bench refers to an environment in which the product under development is tested with the aid of a collection of testing tools. Often, though not always, the suite of testing tools is designed specifically for the product under test.

A test bench or testing workbench has four components:

1. Input: The entrance criteria or deliverables needed to perform work,
2. Procedures to do: The tasks or processes that will transform the input into the output,
3. Procedures to check: The processes that determine that the output meets the standards,
4. Output: The exit criteria or deliverables produced from the workbench.

An example of a software test bench

The tools used to automate the testing process in a test bench perform the following functions:

Test manager: manages the running of program tests; keeps track of test data, expected results and program facilities tested.

Test data generator: generates test data for the program to be tested.

Oracle: generates predictions of the expected test results; the oracle may be either previous program versions or prototype systems.

File comparator: compares the results of the program tests with previous test results and records any differences in a document.

Report generator: provides report definition and generation facilities for the test results.

Dynamic analyzer: adds code to a program to count the number of times each statement has been executed. It generates an execution profile for the statements to show the number of times they are executed in the program run.

Simulator: simulates the testing environment where the software product is to be used.

References

Test case

A **test case** in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. The mechanism for determining whether a software program or system has passed or failed such a test is known as a **test oracle**. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is considered sufficiently scrutinized to be released. Test cases are often referred to as **test scripts**, particularly when written. Written test cases are usually collected into test suites.

Formal test cases

In order to fully test that all the requirements of an application are met, there must be at least two test cases for each requirement: one positive test and one negative test. If a requirement has sub-requirements, each sub-requirement must have at least two test cases. Keeping track of the link between the requirement and the test is frequently done using a traceability matrix. Written test cases should include a description of the functionality to be tested, and the preparation required to ensure that the test can be conducted.

A formal written test-case is characterized by a known input and by an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a postcondition.

Informal test cases

For applications or systems without formal requirements, test cases can be written based on the accepted normal operation of programs of a similar class. In some schools of testing, test cases are not written at all but the activities and results are reported after the tests have been run.

In scenario testing, hypothetical stories are used to help the tester think through a complex problem or system. These scenarios are usually not written down in any detail. They can be as simple as a diagram for a testing environment or they could be a description written in prose. The ideal scenario test is a story that is motivating, credible, complex, and easy to evaluate. They are usually different from test cases in that test cases are single steps while scenarios cover a number of steps of the key.

Typical written test case format

A test case is usually a single step, or occasionally a sequence of steps, to test the correct behaviour/functionalities, features of an application. An expected result or expected outcome is usually given.

Additional information that may be included:

- test case ID
- test case description
- test step or order of execution number
- related requirement(s)
- depth
- test category
- author
- check boxes for whether the test is automatable and has been automated.

Additional fields that may be included and completed when the tests are executed:

- pass/fail
- remarks

Larger test cases may also contain prerequisite states or steps, and descriptions.

A written test case should also contain a place for the actual result.

These steps can be stored in a word processor document, spreadsheet, database or other common repository.

In a database system, you may also be able to see past test results and who generated the results and the system configuration used to generate those results. These past results would usually be stored in a separate table.

Test suites often also contain

- Test summary
- Configuration

Besides a description of the functionality to be tested, and the preparation required to ensure that the test can be conducted, the most time consuming part in the test case is creating the tests and modifying them when the system changes.

Under special circumstances, there could be a need to run the test, produce results, and then a team of experts would evaluate if the results can be considered as a pass. This happens often on new products' performance number determination. The first test is taken as the base line for subsequent test / product release cycles.

Acceptance tests, which use a variation of a written test case, are commonly performed by a group of end-users or clients of the system to ensure the developed system meets the requirements specified or the contract. User acceptance tests are differentiated by the inclusion of happy path or positive test cases to the almost complete exclusion of negative test cases.

References

External links

- Writing Software Security Test Cases - Putting security test cases into your test plan (<http://www.qasec.com/cycle/securitytestcases.shtml>) by Robert Auger

Test data

Test Data are data which have been specifically identified for use in tests, typically of a computer program.

Some data may be used in a confirmatory way, typically to verify that a given set of input to a given function produces some expected result. Other data may be used in order to challenge the ability of the program to respond to unusual, extreme, exceptional, or unexpected input.

Test data may be produced in a focused or systematic way (as is typically the case in domain testing), or by using other, less-focused approaches (as is typically the case in high-volume randomized automated tests). Test data may be produced by the tester, or by a program or function that aids the tester. Test data may be recorded for re-use, or used once and then forgotten.

Domain testing is a family of test techniques that focus on the test data. This might include identifying common or critical inputs, representatives of a particular equivalence class model, values that might appear at the boundaries between one equivalence class and another, outrageous values that should be rejected by the program, combinations of inputs, or inputs that might drive the product towards a particular set of outputs.

References

- "The evaluation of program-based software test data adequacy criteria" ^[1], E. J. Weyuker, Communications of the ACM (abstract and references)
- Free online tool platform for test data generation <http://www.testersdesk.com> ^[2]
- GEDIS Studio is an advanced workbench for generating realistic test data. Community, Pro and Ents versions are available. ^[3]

References

[1] <http://portal.acm.org/citation.cfm?id=62963>

[2] <http://www.testersdesk.com>

[3] <http://www.genielog.com>

Test design

In software engineering, **test design** is the act of creating and writing test suites for testing a software.

Definition

Test design could require all or one of:

- knowledge of the software, and the business area it operates on,
- knowledge of the functionality being tested,
- knowledge of testing techniques and heuristics.
- planning skills to schedule in which order the test cases should be designed, given the effort, time and cost needed or the consequences for the most important and/or risky features.^[1]

Well designed test suites will provide for an efficient testing. The test suite will have just enough test cases to test the system, but no more. This way, there is no time lost in writing redundant test cases that would unnecessarily consume time each time they are executed. In addition, the test suite will not contain brittle or ambiguous test cases.

Automatic test design

Entire test suites or test cases exposing real bugs can be automatically generated by software using model checking or symbolic execution.^[2] Model checking can ensure all the paths of a simple program are exercised, while symbolic execution can detect bugs and generate a test case that will expose the bug when the software is run using this test case.

However, as good as automatic test design can be, it is not appropriate for all circumstances. If the complexity becomes too high, then human test design must come into play as it is far more flexible and it can concentrate on generating higher level test suites.

References

- [1] *A Practitioner's Guide to Software Test Design* (<http://www.amazon.com/Practitioners-Guide-Software-Test-Design/dp/158053791X>), by Lee Copeland, January 2004
- [2] *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs* (<http://www.doc.ic.ac.uk/~cristic/papers/klee-osdi-08.pdf>), by Cristian Cadar, Daniel Dunbar, Dawson Engler of Stanford University,

Test Double

In Computer Programming and Computer Science, especially in Object-oriented programming, programmers and developers employ a technique called, "Automated unit testing" to enhance the quality of the software. Frequently, the final release software consists of a complex set of objects or procedures interacting together to create the final result. In Automated unit testing, it may be necessary to use objects or procedures that look and behave like their release-intended counterparts, but are actually simplified versions that reduce the complexity and facilitate testing. A **Test Double** is a generic (meta) term used for these objects or procedures.

Types of Test Doubles

Gerard Meszaros^[1] identified several different terms for what he calls, "Test Doubles." Using his vocabulary, there are at least five types of Test Doubles:

- Test Stub (used for providing the tested code with "indirect input")
- Mock Object (used for verifying "indirect output" of the tested code, by first defining the expectations before the tested code is executed)
- Test Spy (used for verifying "indirect output" of the tested code, by asserting the expectations afterwards, without having defined the expectations before the tested code is executed)
- Fake Object (used as a simpler implementation, e.g. using an in-memory database in the tests instead of doing real database access)
- Dummy Object (used when a parameter is needed for the tested method but without actually needing to use the parameter)

While there is no open standard for **Test Double** and the various types, there is momentum for continued use of these terms in this manner. Martin Fowler used these terms in his article, *Mocks Aren't Stubs*^[2] referring to Meszaros' book. Microsoft also used the same terms and definitions in an article titled, *Exploring The Continuum Of Test Doubles*.^[3]

References

- [1] Meszaros, Gerard (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley. ISBN 978-0131495050.
- [2] Fowler, Martin (2007). "Mocks Aren't Stubs" (<http://martinfowler.com/articles/mocksArentStubs.html>). . Retrieved 2010-12-29.
- [3] Seemann, Mark (2007). "Exploring The Continuum Of Test Doubles" (<http://msdn.microsoft.com/en-us/magazine/cc163358.aspx>). . Retrieved 2010-12-29.

External links

- <http://xunitpatterns.com/Test%20Double.html> (<http://xunitpatterns.com/Test Double.html>)
- <http://xunitpatterns.com/Test%20Double%20Patterns.html> (<http://xunitpatterns.com/Test Double Patterns.html>)
- <http://www.martinfowler.com/bliki/TestDouble.html> (<http://www.martinfowler.com/bliki/TestDouble.html>)

Test effort

In software development, **test effort** refers to the expenses for (still to come) tests. There is a relation with test costs and failure costs (direct, indirect, costs for fault correction). Some factors which influence test effort are: maturity of the software development process, quality and testability of the testobject, test infrastructure, skills of staff members, quality goals and test strategy.

Methods for estimation of the test effort

To analyse all factors is difficult, because most of the factors influence each other. Following approaches can be used for the estimation: top-down estimation and bottom-up estimation. The top-down techniques are formula based and they are relative to the expenses for development: Function Point Analysis (FPA) and Test Point Analysis (TPA) amongst others. Bottom-up techniques are based on detailed information and involve often experts. The following techniques belong here: Work Breakdown Structure (WBS) and Wide Band Delphi (WBD).

We can also use the following techniques for estimating the test effort -

- Conversion of software size into person hours of effort directly using a conversion factor. For example, we assign 2 person hours of testing effort per one Function Point of software size or 4 person hours of testing effort per one use case point or 3 person hours of testing effort per one Software Size Unit
- Conversion of software size into testing project size such as Test Points or Software Test Units using a conversion factor and then convert testing project size into effort
- Compute testing project size using Test Points of Software Test Units. Methodology for deriving the testing project size in Test Points is not well documented. However, methodology for deriving Software Test Units is defined in a paper by Murali Chemuturi
- We can also derive software testing project size and effort using Delphi Technique or Analogy Based Estimation technique.

Test efforts from literature

In literature test efforts relative to total costs are between 20% and 70%. These values are amongst others dependent from the project specific conditions. When looking for the test effort in the single phases of the test process, these are diversely distributed: with about 40% for test specification and test execution each.

References

- Andreas Spillner, Tilo Linz, Hans Schäfer. (2006). *Software Testing Foundations - A Study Guide for the Certified Tester Exam - Foundation Level - ISTQB compliant*, 1st print. dpunkt.verlag GmbH, Heidelberg, Germany. ISBN 3-89864-363-8.
- Erik van Veenendaal (Hrsg. und Mitautor): *The Testing Practitioner*. 3. Auflage. UTN Publishers, CN Den Bosch, Niederlande 2005, ISBN 90-72194-65-9.
- Thomas Müller (chair), Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhäjärvi, Geoff Thompson and Erik van Veendental. (2005). *Certified Tester - Foundation Level Syllabus - Version 2005*, International Software Testing Qualifications Board (ISTQB), Möhrendorf, Germany. (PDF; 0,424 MB ^[1]).
- Andreas Spillner, Tilo Linz, Thomas Roßner, Mario Winter: *Praxiswissen Softwaretest - Testmanagement: Aus- und Weiterbildung zum Certified Tester: Advanced Level nach ISTQB-Standard*. 1. Auflage. dpunkt.verlag GmbH, Heidelberg 2006, ISBN 3-89864-275-5.

External links

- Wide Band Delphi ^[2]
- Test Effort Estimation ^[3]

References

- [1] <http://www.istqb.org/downloads/syllabi/SyllabusFoundation2005.pdf>
[2] <http://tech.willeke.com/Programing/Guidelines/GL-10.htm>
[3] <http://www.chemuturi.com/Test%20Effort%20Estimation.pdf>

Test execution engine

A **test execution engine** is a type of software used to test software, hardware or complete systems.

Synonyms of test execution engine:

- Test executive
- Test manager

A test execution engine may appear in two forms:

- Module of a test software suite (test bench) or an integrated development environment
- Stand-alone application software

Concept

The test execution engine does not carry any information about the tested product. Only the test specification and the test data carries information about the tested product.

The test specification is software. Test specification is sometimes referred to as test sequence, which consists of test steps.

The test specification should be stored in the test repository in a text format (such as source code). Test data is sometimes generated by some test data generator tool. Test data can be stored in binary or text files. Test data should also be stored in the test repository together with the test specification.

Test specification is selected, loaded and executed by the test execution engine similarly, as application software is selected, loaded and executed by operation systems. The test execution engine should not operate on the tested object directly, but through plug-in modules similarly as an application software accesses devices through drivers which are installed on the operation system.

The difference between the concept of test execution engine and operation system is that the test execution engine monitors, presents and stores the status, results, time stamp, length and other information for every Test Step of a Test Sequence, but typically an operation system does not perform such profiling of a software execution.

Reasons for using a test execution engine:

- Test results are stored and can be viewed in a uniform way, independent of the type of the test
 - Easier to keep track of the changes
 - Easier to reuse components developed for testing
-

Functions

Main functions of a test execution engine:

- Select a test type to execute. Selection can be automatic or manual.
- Load the specification of the selected test type by opening a file from the local file system or downloading it from a Server, depending on where the test repository is stored.
- Execute the test through the use of testing tools (SW test) or instruments (HW test), while showing the progress and accepting control from the operator (for example to Abort)
- Present the outcome (such as Passed, Failed or Aborted) of test Steps and the complete Sequence to the operator
- Store the Test Results in report files

An advanced test execution engine may have additional functions, such as:

- Store the test results in a Database
- Load test result back from the Database
- Present the test results as raw data.
- Present the test results in a processed format. (Statistics)
- Authenticate the operators.

Advanced functions of the test execution engine maybe less important for software testing, but these advanced features could be essential when executing hardware/system tests.

Operations types

A test execution engine by executing a test specification, it may perform different types of operations on the product, such as:

- Verification
- Calibration
- Programming
 - Downloading firmware to the product's nonvolatile memory (Flash)
 - Personalization: programming with unique parameters, like a serial number or a MAC address

If the subject is a software, verification is the only possible operation.

Implementation Examples

Proprietary

Software test:

- IBM's IBM Rational Quality Manager ^[1]

Hardware or system test:

- National Instruments' TestStand ^[2] - Test Management Software
- Hiatronics' Hiatronic Development Suite ^[3] - Test Stand Content Management System
- Geotest's ATEasy ^[4] - Rapid Application Development Framework

Open Source

Hardware or system test:

- JTStand^[5] - Scripting Environment for Data Collection

Choosing a Test execution engine

TBD

References

- [1] <http://www-01.ibm.com/software/awdtools/rqm/standard/>
- [2] <http://www.ni.com/teststand/>
- [3] <http://www.hiatronics.com/>
- [4] <http://www.geotestinc.com/Product.aspx?model=ATEasy/>
- [5] <http://www.jtstand.com/>

Test harness

In software testing, a **test harness** or **automated test framework** is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs. It has two main parts: the Test execution engine and the Test script repository.

Test harnesses allow for the automation of tests. They can call functions with supplied parameters and print out and compare the results to the desired value. The test harness is a hook to the developed code, which can be tested using an automation framework.

A test harness should allow specific tests to run (this helps in optimising), orchestrate a runtime environment, and provide a capability to analyse results.

The typical objectives of a test harness are to:

- Automate the testing process.
- Execute test suites of test cases.
- Generate associated test reports.

A test harness **may** provide some of the following benefits:

- Increased productivity due to automation of the testing process.
- Increased probability that regression testing will occur.
- Increased quality of software components and application.
- Ensure that subsequent test runs are exact duplicates of previous ones.
- Testing can occur at times that the office is not staffed (ie. at night)
- A test script may include conditions and/or uses that are otherwise difficult to simulate (load, for example)

An alternative definition of a test harness is software constructed to facilitate integration testing. Where test stubs are typically components of the application under development and are replaced by working component as the application is developed (top-down design), test harnesses are external to the application being tested and simulate services or functionality not available in a test environment. For example, if you're building an application that needs to interface with an application on a mainframe computer but none is available during development, a test harness maybe built to use as a substitute. A test harness maybe part of a project deliverable. It's kept outside of the application source code and maybe reused on multiple projects. Because a test harness simulates application functionality - it has no knowledge of test suites, test cases or test reports. Those things are provided by a testing framework and associated automated testing tools.

Test management

Test management is the activity of managing some tests. A test management tool is software used to manage tests (automated or manual) that have been previously specified. It is often associated with automation software. Test management tools often include requirement and/or specification management modules that allow automatic generate a requirement test matrix (RTM), which is one of the main metrics to indicate functional coverage of a system under test (SUT).

Creating tests definitions in a database

Test definition includes: test plan, association with product requirements and specifications. Eventually, some relationship can be set between tests so that precedences can be established. i.e. if test A is parent of test B and if test A is failing, then it may be useless to perform test B. Tests should also be associated with priorities. Every change on a test must be versioned so that the QA team has a comprehensive view of the history of the test.

Preparing test campaigns

This includes building some bundles of test cases and execute them (or scheduling their execution). Execution can be either manual or automatic.

Manual execution

The user will have to perform all the test steps manually and inform the system of the result. Some test management tools includes a framework to interface the user with the test plan to facilitate this task.

Automatic execution

There are a numerous way of implementing automated tests. Automatic execution requires the test management tool to be compatible with the tests themselves. To do so, test management tools may propose proprietary automation frameworks or APIs to interface with third-party or proprietary automated tests.

Generating reports and metrics

The ultimate goal of test management tools is to deliver sensitive metrics that will help the QA manager in evaluating the quality of the system under test before releasing. Metrics are generally presented as graphics and tables indicating success rates, progression/regression and much other sensitive data.

Managing bugs

Eventually, test management tools can integrate bug tracking features or at least interface with well-known dedicated bug tracking solutions (such as Bugzilla or Mantis) efficiently link a test failure with a bug.

Planning test activities

Test management tools may also integrate (or interface with third-party) project management functionalities to help the QA manager planning activities ahead of time.

Test management tools

There are several commercial and open source test management tools available in the market today. Some of the popular tools include HP Quality Center, QMetry, PractiTest, IBM Rational, Testlink, Testopia, Testuff, TOSCA, etc. Some of these tools need to be installed in-house, while some can be accessed as SaaS.

External links

- Testing FAQs ^[1]

References

[1] <http://www.testingfaqs.org/t-management.html>

Test Management Approach

Test Management Approach (TMap) is a software testing methodology. TMap is a method which combines insights on how to test and what to manage, as well as techniques for the individual test consultant.

Introduction

History

The first method was created in 1995 and written by Martin Pol, Ruud Teunissen en Erik van Veenendaal. At the end of 2006 a new version was published called *TMap Next* written by other authors (Tim Koomen, Michiel Vroon, Leo van der Aalst & Bart Broekman). The main reason for this new version was the aim to create a more process focussed description of the test process and put more emphasis on the business objectives as a guidance for the testing process.

TMap was created by the Dutch division of Sogeti which is part of Capgemini.

Although TMap is a Dutch product by origin, the method has been translated into French, German and English.

The importance of testing

There are risks involved in changes. Introducing new information systems is a major change for a lot of organisations and it is wise to manage these risks. This is called Enterprise risk management.

The essentials of TMap

TMap Next has 4 pillars:

Business Driven Test Management

(**BDTM**). The test manager can manage the process on 4 aspects (Time, Costs, Risks, and Results).

Toolbox

TMap has a toolbox which provide the techniques to perform the method;

Structure

TMap Next has phases that each test has to go through:

- Plan;
- Preparation;
- Specification;
- Execution;
- Evaluation.

and the two extra phases:

- Infrastructure;
- Management.

Flexible

TMap allows for adaptation to the environment, including agile and scrum.

Master test plan

Planning

In this phase a risk analysis of the product is carried out, and a test strategy is developed. The budget and test plans are made. Choices are made about the products to be delivered, the test infrastructure and the test organisation. The master test plan usually has to be signed off by the business (client).

Test management

In the master test plan the test process controls are specified.

Testing during development

Testing can be done at the end of the process where the end-product is tested against the requirements, or it can be done in an earlier phase, during development. During development, what can be tested are the available elements. What can be tested depends on the Software testability. Testing during the development phase is the review of documentation, and the testing of small parts of the system as soon as they are ready for testing. It is partly Static testing and White-box testing. Examples are: test driven development, pair programming, code review, Continuous integration and application integration. In Agile testing testing is carried out early in the process.

System and Acceptance testing

Although System testing and Acceptance testing are different phases they do have a lot of things in common.

Supporting processes

The supporting processes are:

Test environment

In the test plan the test environment is described.

Test tools

Test tools can be used.

Test professionals

The selection of Test professionals is done early in the process.

Products and Tools

Product risk analysis

A Risk analysis can be made.

Quality

The Software quality control

Budgeting

A budget is always important.

Test results, issues, bugs, problems and show-stoppers

Test results should be documented. This can be done in a simple word document, a spreadsheet, a database or even using specialized applications to manage the findings. It should be clear at any point how many test cases or test scripts are run, how many bugs are found and how many of them are still open. In the beginning of the test process the number of discovered bugs, issues, problems and show-stoppers will grow. They are of course reported back to the developers, who will try to resolve the problems after which they have to be retested, resulting in a diminishing number of open issues and at some point a growing feeling of confidence in the new system.

Metrics

The Performance metrics are used for controlling the process.

Test design

A test design is made after the planning. Subjects are: Create, read, update and delete and Boundary value analysis.

Test Methods

Tmap uses and describes the following test methods.

- Decision tree test
- Data combination test
- All-pairs testing
- Error guessing
- Exploratory testing
- Real life test
- Semantic test
- Use case test

Audit or Review

To speed up and improve the total test process it is good practise not to wait until everything is ready and then test the end product, but to review intermediate products (documentation) and Audit the process as well. All intermediate products can be reviewed. This is called Static testing. Techniques used in this phase are:

- checklists
- Review
- Walkthrough

Intermediate products to test are the: requirements, system design, test strategy, test plan, test scripts, unit test results, prototype.

The results of the formal audits or reviews have to be documented, reported to the project manager and discussed (Feedback) with the authors / developers. This can lead to changes, changes to the documents / products, the process or the people. More informal reviews are also possible, were colleagues or peers are involved.

Test roles

TMap has *three test roles*:

- Testmanager;
- Test Coordinator;
- Tester.

Further reading

- **TMap Next: For Result-driven Testing** (2006)

Tim Koomen, Leo van der Aalst, Bart Broekman, Michiel Vroon, Rob Baarda

ISBN: 9072194802 ^[1]

- * **Software Testing: A guide to the TMap Approach** (2001)

Martin Pol, Ruud Teunissen, Erik van Veenendaal

ISBN 0201745712 ^[2]

External links

- **TMap Next** - Home page ^[3]
- **TMap Next, the test standard** - Webinar Recording ^[4]

References

- [1] <http://www.amazon.com/exec/obidos/ASIN/9072194802/>
- [2] <http://www.amazon.com/exec/obidos/ASIN/0201745712/>
- [3] <http://eng.tmap.net/Home/>
- [4] <http://www.blog.sogeti.ie/2009/09/webinar-tmap-next-test-standard-16-sep.html>

Test plan

A **test plan** is a document detailing a systematic approach to testing a system such as a machine or software. The plan typically contains a detailed understanding of what the eventual workflow will be.

Test plans

A test plan documents the strategy that will be used to verify and ensure that a product or system meets its design specifications and other requirements. A test plan is usually prepared by or with significant input from Test Engineers.

Depending on the product and the responsibility of the organization to which the test plan applies, a test plan may include one or more of the following:

- *Design Verification or Compliance test* - to be performed during the development or approval stages of the product, typically on a small sample of units.
- *Manufacturing or Production test* - to be performed during preparation or assembly of the product in an ongoing manner for purposes of performance verification and quality control.
- *Acceptance or Commissioning test* - to be performed at the time of delivery or installation of the product.
- *Service and Repair test* - to be performed as required over the service life of the product.
- *Regression test* - to be performed on an existing operational product, to verify that existing functionality didn't get broken when other aspects of the environment are changed (e.g., upgrading the platform on which an existing application runs).

A complex system may have a high level test plan to address the overall requirements and supporting test plans to address the design details of subsystems and components.

Test plan document formats can be as varied as the products and organizations to which they apply. There are three major elements that should be described in the test plan: Test Coverage, Test Methods, and Test Responsibilities. These are also used in a formal test strategy.

Test coverage in the test plan states what requirements will be verified during what stages of the product life. Test Coverage is derived from design specifications and other requirements, such as safety standards or regulatory codes, where each requirement or specification of the design ideally will have one or more corresponding means of verification. Test coverage for different product life stages may overlap, but will not necessarily be exactly the same for all stages. For example, some requirements may be verified during Design Verification test, but not repeated during Acceptance test. Test coverage also feeds back into the design process, since the product may have to be designed to allow test access (see Design For Test).

Test methods in the test plan state how test coverage will be implemented. Test methods may be determined by standards, regulatory agencies, or contractual agreement, or may have to be created new. Test methods also specify

test equipment to be used in the performance of the tests and establish pass/fail criteria. Test methods used to verify hardware design requirements can range from very simple steps, such as visual inspection, to elaborate test procedures that are documented separately.

Test responsibilities include what organizations will perform the test methods and at each stage of the product life. This allows test organizations to plan, acquire or develop test equipment and other resources necessary to implement the test methods for which they are responsible. Test responsibilities also includes, what data will be collected, and how that data will be stored and reported (often referred to as "deliverables"). One outcome of a successful test plan should be a record or report of the verification of all design specifications and requirements as agreed upon by all parties.

IEEE 829 test plan structure

IEEE 829-2008, also known as the 829 Standard for Software Test Documentation, is an IEEE standard that specifies the form of a set of documents for use in defined stages of software testing, each stage potentially producing its own separate type of document.^[1]

- Test plan identifier
- Introduction
- Test items
- Features to be tested
- Features not to be tested
- Approach
- Item pass/fail criteria
- Suspension criteria and resumption requirements
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training needs
- Schedule
- Risks and contingencies
- Approvals

There are also other IEEE documents that suggest what should be contained in a test plan:

- 829-1983 IEEE Standard for Software Test Documentation (superseded by 829-1998)^[2]
 - 829-1998 IEEE Standard for Software Test Documentation (superseded by 829-2008)^[3]
 - 1008-1987 IEEE Standard for Software Unit Testing^[4]
 - 1012-2004 IEEE Standard for Software Verification & Validation Plans^[5]
 - 1059-1993 IEEE Guide for Software Verification & Validation Plans (withdrawn)^[6]
-

References

- [1] IEEE Standard 829-2008 (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4578383)
- [2] IEEE Standard 829-1983 (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=573169)
- [3] IEEE Standard 829-1998 (<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=741968&isnumber=16010>)
- [4] IEEE Standard 1008-1987 (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=27763)
- [5] IEEE Standard 1012-2004 (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1488512)
- [6] IEEE Standard 1059-1993 (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=838043)

External links

- Public domain RUP test plan template at Sourceforge (<http://jdbv.sourceforge.net/RUP.html>) (templates are currently inaccessible but sample documents can be seen here: DBV Samples (<http://jdbv.sourceforge.net/Documentation.html>))
- Test plans and test cases (<http://www.stellman-greene.com/testplan>)

Test script

A **test script** in software testing is a set of instructions that will be performed on the system under test to test that the system functions as expected.

There are various means for executing test scripts.

- Manual testing. These are more commonly called test cases.
- Automated testing
 - Short program written in a programming language used to test part of the functionality of a software system. Test scripts written as a short program can either be written using a special automated functional GUI test tool (such as HP QuickTest Professional, Borland SilkTest, and Rational Robot) or in a well-known programming language (such as C++, C#, Tcl, Expect, Java, PHP, Perl, Powershell, Python, or Ruby).
 - Extensively parameterized short programs a.k.a. Data-driven testing
 - Reusable steps created in a table a.k.a. keyword-driven or table-driven testing.

These last two types are also done in manual testing.

The major advantage of *automated testing* is that tests may be executed continuously without the need for a human intervention. Another advantage over *manual testing* is that it is faster and easily repeatable. Thus, it is worth considering automating tests if they are to be executed several times, for example as part of regression testing.

Disadvantages of automated testing are that automated tests can — like any piece of software — be poorly written or simply break during playback. They also can only examine what they have been programmed to examine. Since most systems are designed with human interaction in mind, it is good practice that a human tests the system at some point. A trained manual tester can notice that the system under test is misbehaving without being prompted or directed however automated tests can only examine what they have been programmed to examine. Therefore, when used in regression testing, manual testers can find new bugs while ensuring that old bugs do not reappear while an automated test can only ensure the latter. That is why mixed testing with automated and manual testing can give very good results, automating what needs to be tested often and can be easily checked by a machine, and using manual testing to do test design to add them to the automated tests suite and to do exploratory testing.

One shouldn't fall into the trap of spending more time automating a test than it would take to simply execute it manually, unless it is planned to be executed several times.

Test strategy

Compare with Test plan.

A **test strategy** is an outline that describes the testing portion of the software development cycle. It is created to inform project managers, testers, and developers about some key issues of the testing process. This includes the testing objective, methods of testing new functions, total time and resources required for the project, and the testing environment.

Test strategies describes how the product risks of the stakeholders are mitigated at the test-level, which types of test are to be performed, and which entry and exit criteria apply. They are created based on development design documents. System design documents are primarily used and occasionally, conceptual design documents may be referred to. Design documents describe the functionality of the software to be enabled in the upcoming release. For every stage of development design, a corresponding test strategy should be created to test the new feature sets.

Test Levels

The test strategy describes the test level to be performed. There are primarily three levels of testing: unit testing, integration testing, and system testing. In most software development organizations, the developers are responsible for unit testing. Individual testers or test teams are responsible for integration and system testing.

Roles and Responsibilities

The roles and responsibilities of test leader, individual testers, project manager are to be clearly defined at a project level in this section. This may not have names associated: but the role has to be very clearly defined.

Testing strategies should be reviewed by the developers. They should also be reviewed by test leads for all levels of testing to make sure the coverage is complete yet not overlapping. Both the testing manager and the development managers should approve the test strategy before testing can begin.

Environment Requirements

Environment requirements are an important part of the test strategy. It describes what operating systems are used for testing. It also clearly informs the necessary OS patch levels and security updates required. For example, a certain test plan may require Windows XP Service Pack 3 to be installed as a prerequisite for testing.

Testing Tools

There are two methods used in executing test cases: manual and automated. Depending on the nature of the testing, it is usually the case that a combination of manual and automated testing is the best testing method. Planner should find the appropriate automation tool to reduce total testing time.

Risks and Mitigation

Any risks that will affect the testing process must be listed along with the mitigation. By documenting a risk, its occurrence can be anticipated well ahead of time. Proactive action may be taken to prevent it from occurring, or to mitigate its damage. Sample risks are dependency of completion of coding done by sub-contractors, or capability of testing tools.

Test Schedule

A *test plan* should make an estimation of how long it will take to complete the testing phase. There are many requirements to complete testing phases. First, testers have to execute all test cases at least once. Furthermore, if a defect was found, the developers will need to fix the problem. The testers should then re-test the failed test case until it is functioning correctly. Last but not the least, the tester need to conduct regression testing towards the end of the cycle to make sure the developers did not accidentally break parts of the software while fixing another part. This can occur on test cases that were previously functioning properly.

The test schedule should also document the number of testers available for testing. If possible, assign test cases to each tester.

It is often difficult to make an accurate approximation of the test schedule since the testing phase involves many uncertainties. Planners should take into account the extra time needed to accommodate contingent issues. One way to make this approximation is to look at the time needed by the previous releases of the software. If the software is new, multiplying the initial testing schedule approximation by two is a good way to start.

Regression Test Approach

When a particular problem is identified, the programs will be debugged and the fix will be done to the program. To make sure that the fix works, the program will be tested again for that criteria. Regression test will make sure that one fix does not create some other problems in that program or in any other interface. So, a set of related test cases may have to be repeated again, to make sure that nothing else is affected by a particular fix. How this is going to be carried out must be elaborated in this section. In some companies, whenever there is a fix in one unit, all unit test cases for that unit will be repeated, to achieve a higher level of quality.

Test Groups

From the list of requirements, we can identify related areas, whose functionality is similar. These areas are the test groups. For example, in a railway reservation system, anything related to ticket booking is a functional group; anything related with report generation is a functional group. Same way, we have to identify the test groups based on the functionality aspect.

Test Priorities

Among test cases, we need to establish priorities. While testing software projects, certain test cases will be treated as the most important ones and if they fail, the product cannot be released. Some other test cases may be treated like cosmetic and if they fail, we can release the product without much compromise on the functionality. This priority levels must be clearly stated. These may be mapped to the test groups also.

Test Status Collections and Reporting

When test cases are executed, the test leader and the project manager must know, where exactly the project stands in terms of testing activities. To know where the project stands, the inputs from the individual testers must come to the test leader. This will include, what test cases are executed, how long it took, how many test cases passed, how many failed, and how many are not executable. Also, how often the project collects the status is to be clearly stated. Some projects will have a practice of collecting the status on a daily basis or weekly basis.

Test Records Maintenance

When the test cases are executed, we need to keep track of the execution details like when it is executed, who did it, how long it took, what is the result etc. This data must be available to the test leader and the project manager, along with all the team members, in a central location. This may be stored in a specific directory in a central server and the document must say clearly about the locations and the directories. The naming convention for the documents and files must also be mentioned.

Requirements traceability matrix

Ideally, the software must completely satisfy the set of requirements. From design, each requirement must be addressed in every single document in the software process. The documents include the HLD, LLD, source codes, unit test cases, integration test cases and the system test cases. In a requirements traceability matrix, the rows will have the requirements. The columns represent each document. Intersecting cells are marked when a document addresses a particular requirement with information related to the requirement ID in the document. Ideally, if every requirement is addressed in every single document, all the individual cells have valid section ids or names filled in. Then we know that every requirement is addressed. If any cells are empty, it represents that a requirement has not been correctly addressed.

Test Summary

The senior management may like to have test summary on a weekly or monthly basis. If the project is very critical, they may need it even on daily basis. This section must address what kind of test summary reports will be produced for the senior management along with the frequency.

The test strategy must give a clear vision of what the testing team will do for the whole project for the entire duration. This document will/may be presented to the client also, if needed. The person, who prepares this document, must be functionally strong in the product domain, with very good experience, as this is the document that is going to drive the entire team for the testing activities. Test strategy must be clearly explained to the testing team members right at the beginning of the project.

References

- Ammann, Paul and Offutt, Jeff. Introduction to software testing. New York: Cambridge University Press, 2008
- Dasso, Aristides. Verification, validation and testing in software engineering. Hershey, PA: Idea Group Pub., 2007

Test stubs

In computer science, **test stubs** are programs which simulate the behaviors of software components (or modules) that are the dependent modules of the module being tested.

“Test stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.”^[1]

Test Stubs are mainly used in incremental testing's Top-Down approach. Stubs are software programs which act as a module and give the output as given by an actual product/software.

Example

Consider a software program which queries a database to obtain the sum price total of all products stored in the database. However, the query is slow and consumes a large number of system resources. This reduces the number of test runs per day. Secondly, the tests need to be conducted on values larger than what is currently in the database.

The method (or call) used to perform this is `get_total()`. For testing purposes, the source code in `get_total()` could be temporarily replaced with a simple statement which returned a specific value. This would be a test stub.

There are several testing frameworks available and there is software that can generate **test stubs** based on existing source code and testing requirements.

External links

- <http://xunitpatterns.com/Test%20Stub.html>^[2]

References

[1] Fowler, Martin (2007), *Mocks Aren't Stubs* (Online) (<http://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs>)

[2] <http://xunitpatterns.com/Test%20Stub.html>

Test suite

In software development, a **test suite**, less commonly known as a *validation suite*, is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviours. A test suite often contains detailed instructions or goals for each collection of test cases and information on the system configuration to be used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Collections of test cases are sometimes incorrectly termed a test plan, a test script, or even a test scenario.

Types

Occasionally, test suites are used to group similar test cases together. A system might have a smoke test suite that consists only of smoke tests or a test suite for some specific functionality in the system. It may also contain all tests and signify if a test should be used as a smoke test or for some specific functionality.

An *executable test suite* is a test suite that can be executed by a program. This usually means that a test harness, which is integrated with the suite, exists. The test suite and the test harness together can work on a sufficiently detailed level to correctly communicate with the system under test (SUT).

A test suite for a primality testing subroutine might consist of a list of numbers and their primality (prime or composite), along with a testing subroutine. The testing subroutine would supply each number in the list to the primality tester, and verify that the result of each test is correct.

External links

- The Plum Hall Validation Suite for C/C++ and the C++ Library ^[1], a popular executable Test Suite.

References

[1] <http://www.plumhall.com/suites.html>

Test Template Framework

The Test Template Framework (TTF) is a model-based testing (MBT) framework proposed by Phil Stocks and David Carrington in (Stocks & Carrington 1996). Although the TTF was meant to be notation-independent, the original presentation was made using the Z formal notation. It is one of the few MBT frameworks approaching unit testing.

Introduction

The TTF is a specific proposal of model-based testing (MBT). It considers models to be Z specifications. Each operation within the specification is analyzed to derive or generate abstract test cases. This analysis consists of the following steps:

1. Define the *input space* (IS) of each operation.
2. Derive the *valid input space* (VIS) from the IS of each operation.
3. Apply one or more *testing tactics*^[1], starting from each VIS, to build a *testing tree* for each operation. Testing trees are populated with nodes called *test classes*.
4. *Prune* each of the resulting testing trees.
5. Find one or more *abstract test cases* from each leaf in each testing tree.

One of the main advantages of the TTF is that all of these concepts are expressed in the same notation of the specification, i.e. the Z notation. Hence, the engineer has to know only one notation to perform the analysis down to the generation of abstract test cases.

Important concepts

In this section the main concepts defined by the TTF are described.

Input space

Let Op be a Z operation. Let $x_1 \dots x_n$ be all the input and (non-primed) state variables referenced in Op , and $T_1 \dots T_n$ their corresponding types. The *Input Space* (IS) of Op , written IS_{Op} , is the Z schema box defined by $[x_1 : T_1 \dots x_n : T_n]$.

Valid input space

Let Op be a Z operation. Let $\text{pre } Op$ be the precondition of Op . The *Valid Input Space* (VIS) of Op , written VIS_{Op} , is the Z schema box defined by $[IS_{Op} | \text{pre } Op]$.

Test class

Let Op be a Z operation and let P be any predicate depending on one or more of the variables defined in VIS_{Op} . Then, the Z schema box $[VIS_{Op} | P]$ is a *test class* of Op . Note that this schema is equivalent to $[IS_{Op} | \text{pre } Op \wedge P]$. This observation can be generalized by saying that if C_{Op} is a test class of Op , then the Z schema box defined by $[C_{Op} | P]$ is also a test class of Op . According to this definition the VIS is also a test

class. If C'_{Op} is a test class of Op , then the predicate P in $C'_{Op} == [C_{Op} | P]$ is said to be the *characteristic predicate* of C'_{Op} or C'_{Op} is *characterized* by P .

Test classes are also called test objectives (Utting & Legeard 2007), test templates (Stocks & Carrington 1996) and test specifications.

Testing tactic

In the context of the TTF a *testing tactic*^[1] is a means to partition any test class of any operation. However, some of the testing tactics used in practice actually do not always generate a partition of some test classes.

Some testing tactics originally proposed for the TTF are the following:

- Disjunctive Normal Form (DNF). By applying this tactic the operation is written in Disjunctive Normal Form and the test class is divided in as many test classes as terms are in the resulting operation's predicate. The predicate added to each new test class is the precondition of one of the terms in the operation's predicate.
- Standard Partitions (SP). This tactic uses a predefined partition of some mathematical operator (Stocks 1993). For example, the following is a good partition for expressions of the form $S \spadesuit T$ where \spadesuit is one of \cup , \cap and \setminus (see Set theory).

$S = \emptyset, T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, S \subset T$
$S = \emptyset, T \neq \emptyset$	$S \neq \emptyset, T \neq \emptyset, T \subset S$
$S \neq \emptyset, T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, T = S$
$S \neq \emptyset, T \neq \emptyset, S \cap T = \emptyset$	$S \neq \emptyset, T \neq \emptyset, S \cap T \neq \emptyset, \neg(S \subseteq T), \neg(T \subseteq S), S \neq T$

As can be noticed, standard partitions might change according to how much testing the engineer wants to perform.

- Sub-domain Propagation (SDP). This tactic is applied to expressions containing:
 1. Two or more mathematical operators for which there are already defined standard partitions, or
 2. Mathematical operators which are defined in terms of other mathematical operators.

In any of these cases, the standard partitions of the operators appearing in the expression or in the definition of a complex one, are combined to produce a partition for the expression. If the tactic is applied to the second case, then the resulting partition can be considered as the standard partition for that operator. Stocks and Carrington in (Stocks & Carrington 1996) illustrate this situation with $R \oplus G = (\text{dom } G \not\subseteq R) \cup G$, where $\not\subseteq$ means domain anti-restriction, by giving standard partitions for $\not\subseteq$ and \cup and propagating them to calculate a partition for \oplus .

- Specification Mutation (SM). The first step of this tactic consists in generating a *mutant* of the Z operation. A mutant of a Z operation is similar in concept to a mutant of a program, i.e. it is a modified version of the operation. The modification is introduced by the engineer with the intention of uncovering an error in the implementation. The mutant should be the specification that the engineer guesses the programmer has implemented. Then, the engineer has to calculate the subset of the VIS that yields different results in both specifications. The predicate of this set is used to derive a new test class.

Some other testing tactics that may also be used are the following:

- In Set Extension (ISE). It applies to predicates of the form $expr \in \{expr_1, \dots, expr_n\}$. In this case, it generates n test classes such that a predicate of the form $expr = expr_i$ is added to each of them.
- Mandatory Test Set (MTS). This tactic associates a set of constant values to a VIS' variable and generates as many test classes as elements are in the set. Each test class is characterized by a predicate of the form $var = val$ where var is the name of the variable and val is one of the values of the set.
- Numeric Ranges (NR). This tactic applies only to VIS' variables of type \mathbb{Z} (or its "subtype" \mathbb{N}). It consists in associating a range to a variable and deriving test classes by comparing the variable with the limits of the range in some ways. More formally, let n be a variable of type \mathbb{Z} and let $[i, j]$ be the associated range. Then, the tactic generates the test classes characterized by the following predicates: $n < i, n = i, i < n \wedge n < j, n = j, n > j$.

- Free Type (FT). This tactic generates as many test classes as elements a free (enumerated) type has. In other words, if a model defines type $COLOUR ::= red|blue|green$ and some operation uses c of type $COLOUR$, then by applying this tactic each test class will be divided into three new test classes: one in which c equals red , the other in which c equals $blue$, and the third where c equals $green$.
- Proper Subset of Set Extension (PSSE). This tactic uses the same concept of ISE but applied to set inclusions. PSSE helps to test operations including predicates like $expr \subset \{expr_1, \dots, expr_n\}$. When PSSE is applied it generates $2^n - 1$ test classes where a predicate of the form $expr = A_i$ with $i \in [1, 2^n - 1]$ and $A_i \in \mathbb{P}\{expr_1, \dots, expr_n\} \setminus \{\{expr_1, \dots, expr_n\}\}$, is added to each class. $\{expr_1, \dots, expr_n\}$ is excluded from $\mathbb{P}\{expr_1, \dots, expr_n\}$ because $expr$ is a proper subset of $\{expr_1, \dots, expr_n\}$.
- Subset of Set Extension (SSE). It is identical to PSSE but it applies to predicates of the form $expr \subseteq \{expr_1, \dots, expr_n\}$ in which case it generates 2^n by considering also $\{expr_1, \dots, expr_n\}$.

Testing tree

The application of a testing tactic to the VIS generates some test classes. If some of these test classes are further partitioned by applying one or more testing tactics, a new set of test classes is obtained. This process can continue by applying testing tactics to the test classes generated so far. Evidently, the result of this process can be drawn as a tree with the VIS as the root node, the test classes generated by the first testing tactic as its children, and so on. Furthermore, Stocks and Carrington in (Stocks & Carrington 1996) propose to use the Z notation to build the tree, as follows.

$$VIS == [IS|P]$$

$$TCL_{T_1}^1 == [VIS|P_{T_1}^1]$$

...

$$TCL_{T_1}^n == [VIS|P_{T_1}^n]$$

$$TCL_{T_2}^1 == [TCL_{T_1}^i|P_{T_2}^1]$$

...

$$TCL_{T_2}^m == [TCL_{T_1}^i|P_{T_2}^m]$$

...

$$TCL_{T_3}^1 == [TCL_{T_2}^j|P_{T_3}^1]$$

...

$$TCL_{T_3}^k == [TCL_{T_2}^j|P_{T_3}^k]$$

...

...

...

Pruning testing trees

In general a test class' predicate is a conjunction of two or more predicates. It is likely, then, that some test classes are empty because their predicates are contradictions. These test classes must be pruned from the testing tree because they represent impossible combinations of input values, i.e. no abstract test case can be derived out of them.

Abstract test case

An abstract test case is an element belonging to a test class. The TTF prescribes that abstract test cases should be derived only from the leaves of the testing tree. Abstract test cases can also be written as Z schema boxes. Let Op be some operation, let VIS_{Op} be the VIS of Op , let $x_1 : T_1 \dots x_n : T_n$ be all the variables declared in VIS_{Op} , let C_{Op} be a (leaf) test class of the testing tree associated to Op , let $P_1 \dots P_m$ be the characteristic predicates of each test class from C_{Op} up to VIS_{Op} (by following the edges from child to parent), and let $v_1 : T_1 \dots v_n : T_n$ be n constant values satisfying $P_1 \wedge \dots \wedge P_m$. Then, an abstract test case of C_{Op} is the Z schema box defined by $[C_{Op} | x_1 = v_1 \wedge \dots \wedge x_n = v_n]$.

References

- Stocks, Phil; Carrington, David (1996), "A framework for specification-based testing", *IEEE Transactions on Software Engineering* **22** (11): 777–793.
- Utting, Mark; Legeard, Bruno (2007), *Practical Model-Based Testing: A Tools Approach* (1st ed.), Morgan Kaufmann, ISBN 0123725011.
- Stocks, Phil (1993), *Applying Formal Methods to Software Testing*, Department of Computer Science, University of Queensland, PhD thesis.

Notes

- [1] Stocks and Carrington use the term *testing strategies* in (Stocks & Carrington 1996).

Test Vector Generator

Test Vector Generator or TVG is a term used to describe a program used to automatically generate test data for use in automated testing of software.

External links

- TVG ^[1], a SourceForge project

References

[1] <http://sourceforge.net/projects/tvg/>

Test-driven development

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.^[1]

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999,^[2] but more recently has created more general interest in its own right.^[3]

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.^[4]

Requirements

Test-driven development requires developers to create automated unit tests that define code requirements (immediately) before writing the code itself. The tests contain assertions that are either true or false. Passing the tests confirms correct behavior as developers evolve and refactor the code. Developers often use testing frameworks, such as xUnit, to create and automatically run sets of test cases.

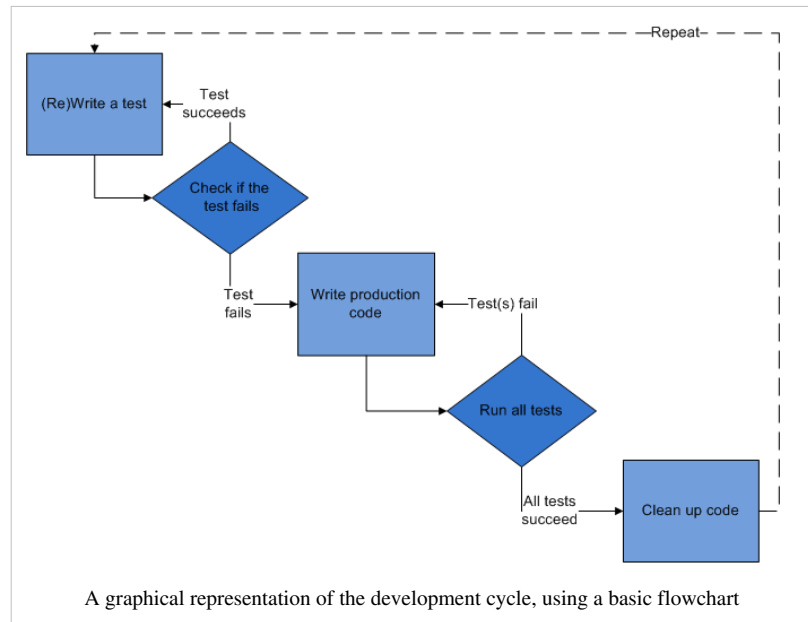
Test-driven development cycle

The following sequence is based on the book *Test-Driven Development by Example*.^[1]

Add a test

In **test-driven development**, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. (If it does not fail, then either the proposed “new” feature already exists or the test is defective.) To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through

use cases and user stories that cover the requirements and exception conditions. This could also imply a variant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.



Run all tests and see if the new one fails

This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. This step also tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be worthless. The new test should also fail for the expected reason. This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

Write some code

The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it.

It is important that the code written is *only* designed to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

Run the automated tests and see them succeed

If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

Refactor code

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code — for example magic numbers or strings that were repeated in both, in order to make the test pass in step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous Integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself,^[3] unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the main program being written.

Development style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You ain't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods.^[1] In *Test-Driven Development by Example*, Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept (such as a design pattern), tests are written that will generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Write the tests first. The tests should be written before the functionality that is being tested. This has been claimed to have two benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature will be written. When writing feature-first code, there is a tendency by developers and the development organisations to push the developer on to the next feature, neglecting testing entirely. The first test might not even compile, at first, because all of the classes and methods it requires may not yet exist. Nevertheless, that first test functions as an executable specification^[5].

First fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has been coined the "test-driven development mantra", known as red/green/refactor where red means *fail* and green is *pass*.

Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the programmer's mental model of the code, boosts confidence and increases productivity.

Advanced practices of test-driven development can lead to Acceptance Test-driven development (ATDD) where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process.^[6] This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target

to satisfy, the acceptance tests, which keeps them continuously focused on what the customer really wants from that user story.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.^[7] Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.^[8]

Programmers using pure TDD on new ("greenfield") projects report they only rarely feel the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.^[9]

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality will be used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, total code implementation time is typically shorter.^[10] Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, in order for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they will detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Vulnerabilities

- Test-driven development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.
- Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.^[11]
- Unit tests created in a test-driven development environment are typically created by the developer who will also write the code that is being tested. The tests may therefore share the same blind spots with the code: If, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify these input parameters. If the developer misinterprets the requirements specification for the module being developed, both the tests and the code will be wrong.
- The high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.
- The tests themselves become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or which are themselves prone to failure, are expensive to maintain. This is especially the case with Fragile Tests.^[12] There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.
- The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original tests become increasingly precious as time goes by. If a poor architecture, a poor design or a poor testing strategy leads to a late change that makes dozens of existing tests fail, it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Code visibility

Test suite code clearly has to be able to access the code it is testing. On the other hand, normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

In object oriented design this still does not provide access to `private` data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access fields that are marked `private`.^[13] Alternatively, an inner class can be used to hold the unit tests so they will have visibility of the enclosing class's members and attributes. In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code. This then means that the released code is not exactly the same as that which is unit tested. The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can then ensure (among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private and protected methods and data anyway. Some argue that it should be sufficient to test any class through its public interface as the private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Others say that crucial aspects of functionality may be

implemented in private methods, and that developing this while testing it indirectly via the public interface only obscures the issue: unit testing is about testing the smallest unit of functionality possible.^{[14] [15]}

Fakes, mocks and integration tests

Unit tests are so named because they each test *one unit* of code. A complex module may have a thousand unit tests and a simple module may have only ten. The tests used for TDD should never cross process boundaries in a program, let alone network connections. Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite. Introducing dependencies on external modules or data also turns *unit tests* into *integration tests*. If one module misbehaves in a chain of interrelated modules, it is not so immediately clear where to look for the cause of the failure.

When code under development relies on a database, a web service, or any other external process or service, enforcing a unit-testable separation is also an opportunity and a driving force to design more modular, more testable and more reusable code.^[16] Two steps are necessary:

1. Whenever external access is going to be needed in the final design, an interface should be defined that describes the access that will be available. See the dependency inversion principle for a discussion of the benefits of doing this regardless of TDD.
2. The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock. Fake objects need do little more than add a message such as “Person object saved” to a trace log, against which a test assertion can be run to verify correct behaviour. Mock objects differ in that they themselves contain test assertions that can make the test fail, for example, if the person's name and other data are not as expected. Fake and mock object methods that return data, ostensibly from a data store or user, can help the test process by always returning the same, realistic data that tests can rely upon. They can also be set into predefined fault modes so that error-handling routines can be developed and reliably tested. Fake services other than data stores may also be useful in TDD: Fake encryption services may not, in fact, encrypt the data passed; fake random number services may always return 1. Fake or mock implementations are examples of dependency injection.

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the “real” implementations of the interfaces discussed above. These tests are quite separate from the TDD unit tests, and are really integration tests. There will be fewer of them, and they need to be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

Integration tests that alter any persistent store or database should always be designed carefully with consideration of the initial and final state of the files or database, even if any test fails. This is often achieved using some combination of the following techniques:

- The `TearDown` method, which is integral to many test frameworks.
- `try...catch...finally` exception handling structures where available.
- Database transactions where a transaction atomically includes perhaps a write, a read and a matching delete operation.
- Taking a “snapshot” of the database before running any tests and rolling back to the snapshot after each test run. This may be automated using a framework such as Ant or NAnt or a continuous integration system such as CruiseControl.
- Initialising the database to a clean state *before* tests, rather than cleaning up *after* them. This may be relevant where cleaning up may make it difficult to diagnose test failures by deleting the final state of the database before detailed diagnosis can be performed.

References

- [1] Beck, K. *Test-Driven Development by Example*, Addison Wesley, 2003
- [2] Lee Copeland (December 2001). "Extreme Programming" (<http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html>). Computerworld. . Retrieved January 11, 2011.
- [3] Newkirk, JW and Vorontsov, AA. *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004.
- [4] Feathers, M. *Working Effectively with Legacy Code*, Prentice Hall, 2004
- [5] http://www.agilesherpa.org/agile_coach/engineering_practices/test_driven_development/
- [6] Koskela, L. "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007
- [7] Erdogmus, Hakan; Morisio, Torchiano. "On the Effectiveness of Test-first Approach to Programming" (<http://nparc.cisti-icist.nrc-cnrc.gc.ca/npsi/ctrl?action=shwart&index=an&req=5763742&lang=en>). *Proceedings of the IEEE Transactions on Software Engineering*, 31(1). January 2005. (NRC 47445). . Retrieved 2008-01-14. "We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive."
- [8] Proffitt, Jacob. "TDD Proven Effective! Or is it?" (<http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx>). . Retrieved 2008-02-21. "So TDD's relationship to quality is problematic at best. Its relationship to productivity is more interesting. I hope there's a follow-up study because the productivity numbers simply don't add up very well to me. There is an undeniable correlation between productivity and the number of tests, but that correlation is actually stronger in the non-TDD group (which had a single outlier compared to roughly half of the TDD group being outside the 95% band)."
- [9] Llopis, Noel (20 February 2005). "Stepping Through the Looking Glass: Test-Driven Game Development (Part 1)" (<http://www.gamesfromwithin.com/articles/0502/000073.html>). Games from Within. . Retrieved 2007-11-01. "Comparing [TDD] to the non-test-driven development approach, you're replacing all the mental checking and debugger stepping with code that verifies that your program does exactly what you intended it to do."
- [10] Müller, Matthias M.; Padberg, Frank. "About the Return on Investment of Test-Driven Development" (<http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf>) (PDF). Universität Karlsruhe, Germany. pp. 6. . Retrieved 2007-11-01.
- [11] Loughran, Steve (November 6, 2006). "Testing" (<http://people.apache.org/~stevel/slides/testing.pdf>) (PDF). HP Laboratories. . Retrieved 2009-08-12.
- [12] "Fragile Tests" ([http://xunitpatterns.com/Fragile Test.html](http://xunitpatterns.com/Fragile%20Test.html)). .
- [13] Burton, Ross (11/12/2003). "Subverting Java Access Protection for Unit Testing" (<http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html>). O'Reilly Media, Inc.. . Retrieved 2009-08-12.
- [14] Newkirk, James (7 June 2004). "Testing Private Methods/Member Variables - Should you or shouldn't you" (<http://blogs.msdn.com/jamesnewkirk/archive/2004/06/07/150361.aspx>). Microsoft Corporation. . Retrieved 2009-08-12.
- [15] Stall, Tim (1 Mar 2005). "How to Test Private and Protected methods in .NET" (<http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx>). CodeProject. . Retrieved 2009-08-12.
- [16] Fowler, Martin (1999). *Refactoring - Improving the design of existing code*. Boston: Addison Wesley Longman, Inc.. ISBN 0-201-48567-2.

External links

- TestDrivenDevelopment on WikiWikiWeb
- Test or spec? Test and spec? Test from spec! (http://www.eiffel.com/general/monthly_column/2004/september.html), by Bertrand Meyer (September 2004)
- Microsoft Visual Studio Team Test from a TDD approach ([http://msdn.microsoft.com/en-us/library/ms379625\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx))
- Write Maintainable Unit Tests That Will Save You Time And Tears (<http://msdn.microsoft.com/en-us/magazine/cc163665.aspx>)
- Improving Application Quality Using Test-Driven Development (TDD) (<http://www.methodsandtools.com/archive/archive.php?id=20>)

Test-Driven Development by Example

Test Driven Development: By Example is a book about a software development technique by Kent Beck.

Beck's concept of test-driven development centers on two basic rules:

1. Never write a single line of code unless you have a failing automated test.
2. Eliminate duplication.

The book illustrates the use of unit testing as part of the methodology, including examples in Java and Python. One section includes using test-driven development to develop a unit testing framework.

References

- *Test Driven Development: By Example, Kent Beck, Addison-Wesley Longman, 2002, ISBN 0321146530, ISBN 978-0321146533*

Test-Driven development helps to achieve re-usability of the code which in turn saves the time and efforts required to write a fresh new code.

Testbed

A **testbed** (also commonly spelled as **test bed** in research publications) is a platform for experimentation of large development projects. Testbeds allow for rigorous, transparent, and replicable testing of scientific theories, computational tools, and new technologies.

The term is used across many disciplines to describe a development environment that is shielded from the hazards of testing in a *live* or production environment. It is a method of testing a particular module (function, class, or library) in an isolated fashion. May be implemented similar to a sandbox, but not necessarily for the purposes of security. A testbed is used as a proof of concept or when a new module is tested apart from the program/system it will later be added to. A skeleton framework is implemented around the module so that the module behaves as if already part of the larger program.

A typical testbed could include software, hardware, and networking components. In software development, the specified hardware and software environment can be set up as a testbed for the application under test. In this context, a testbed is also known as the test environment.

Testbeds are also pages on the Internet where the public is given the opportunity to test CSS or HTML they have created and want to preview the results.

Examples

The Arena web browser was created by the World Wide Web Consortium (W3C) and CERN for testing HTML3, Cascading Style Sheets (CSS), Portable Network Graphics (PNG) and the libwww.^[1] ^[2] Arena was replaced by Amaya to test new web standards ^[3]

The Line Mode browser got a new function to interact with the libwww library as a sample and test application.^[4]

The libwww was also created to test network protocols which are under development or to experiment with new protocols.^[5]

References

- [1] QingLong, Lu. "The Arena Web Browser" (<http://web.archive.org/web/20030228181036/www.yggdrasil.com/Products/Arena/>). Yggdrasil Computing. Archived from the original (<http://www.yggdrasil.com/Products/Arena/>) on 22 February 2003. . Retrieved 6 June 2010.
- [2] "Web working group - Minutes "Navigation, services and interoperability" session" (<http://event.cwi.nl/W4G/WS94/usertools.html>). *World Wide Web Working group*. . Retrieved 9 June 2010.
- [3] "History of the Web" (<http://www.w3c.rl.ac.uk/primers/history/origins.htm>). Oxford Brookes University. 2002. . Retrieved 10 July 2010.
- [4] Nielsen, Henrik Frystyk (4 May 1998). "WWW - The Libwww Line Mode Browser" (<http://www.w3.org/LineMode/>). World Wide Web Consortium. . Retrieved 9 June 2010.
- [5] "libwww" (<http://www.ros.org/wiki/libwww>). ROS. . Retrieved 2 June 2010.

External links

- PlanetLab Europe, the European portion of the publicly available PlanetLab testbed (<http://www.planet-lab.eu/>)
- CMU's eRulemaking Testbed (<http://erulemaking.cs.cmu.edu/Data.html>)
- US National Science Foundation GENI - Global Environment for Network Innovations Initiative
- Helsinki Testbed (meteorology) (<http://testbed.fmi.fi/>)
- Collaborative Adaptive Sensing of the Atmosphere (CASA) IP1 test bed (http://www.casa.umass.edu/main/research/technicalintegration/integrated_project_1/)

Tester driven development

Tester driven development is an anti-pattern in software development. It should not be confused with test driven development. It refers to any software development project where the software testing phase is too long. The testing phase is so long that the requirements may change radically during software testing. New or changed requirements often appear as bug reports. Bug tracking software usually lacks support for handling requirements. As a result of this nobody really knows what the system requirements are.

Projects that are developed using this anti-pattern often suffer from being extremely late. Another common problem is poor code quality.

Common causes for projects ending up being run this way are often:

- The testing phase started too early;
- Incomplete requirements;
- Inexperienced testers;
- Inexperienced developers;
- Poor project management.

Things get worse when the testers realise that they don't know what the requirements are and therefore don't know how to test any particular code changes. The onus then falls on the developers of individual changes to write their own test cases and they are happy to do so because their own tests normally pass and their performance measurements improve. Project leaders are also delighted by the rapid reduction in the number of open change requests.

Tester forum

The UK **Tester Forum** is an online resource and a series of quarterly meetings in Central London on the last Wednesday of January, April, July and October. The Forum was invented and is currently hosted by Paul Gerrard (consultant) of Gerrard Consulting.

The forums cover three main topic areas:

- Test Management
- Performance Testing
- Test Automation

The Forums are open to anyone on the planet, but meetings take place in London UK every quarter.

Test Management Forum

The Test Management Forum is aimed at senior testing practitioners and managers. It aims to be:

- Forward looking, focusing on trends, futures, the 'big issues' in test management

Geared towards networking - bringing like-minded test managers together

- To be influential - to have a certain gravitas, authority and influence over the testing industry - without being pompous!
- A tools-free zone with no sales-pitches or vendor preferences

Performance Testing Forum

The Performance Testing Forum is aimed at performance test professionals. It aims to be:

- Forward looking, focusing on trends, futures, the 'big issues' in performance testing

Geared towards networking - bringing like-minded performance testers together

- To inform - to gather experts in the field to discuss techniques, approaches and tools and disseminate good ideas.

Organisation

The quarterly Forums in April, July and October are sponsored events, open to all and free to attend.

The January Forum is special and we call it the Summit. The annual Summit is a sponsored event, open to all and for a remarkably low fee.

The Forums and Summit are organised and hosted by Gerrard Consulting.

Contributions and content are invited from practitioners.

It is non-bureaucratic, driven by participants.

External links

- Organizational website ^[1]

References

[1] <http://uktmf.com>

Testing as a service

Testing as a Service (TaaS), typically pronounced 'tass') is a model of software testing whereby a provider undertakes the activity of software testing applications/solutions for customers as a service on demand. Testing as a Service involves the on-demand test execution of well-defined suites of test material, generally on an outsourced basis. The execution can be performed either on client site or remotely from the outsourced providers test lab/facilities.

Aims and objectives

One of the main objectives of TaaS is to allow the organisation to focus on core business activities while keeping costs down, thus allowing them to address the current (2008/9) trend of reduced IT budgets while not getting distracted on non-core competencies.^[1]

Key characteristics and delivery

In order for TaaS to work effectively it should be delivered on either a fixed price or known cost basis with clearly defined schedules of work, this will allow organisations to deal with the high levels of change that might be inherent in their solutions/organisations while allowing them to smooth any resource peaks and troughs.^[2]

There is no clear evidence to support which is the most effective delivery model for TaaS, whether that be onshore, near shore or indeed offshore, it is however clear that for most organisations speed of testing is important and so the use of people versus technology must be considered very carefully.

Implementation

While it is not true for every system under test (SUT) or application under test (AUT), it is true that in certain circumstances the use of test automation can massively increase the speed and efficiency of test execution.^[3] So any TaaS offering, if delivered using test automation will allow organisations to benefit from reduced time to market without suffering the learning curve and headaches involved in undertaking test automation.^[4]

References

- [1] Value versus cost: governing IT on a reduced budget, Retrieved, 2010/01/04 <http://www.computerweekly.com/Articles/2002/02/08/185114/value-versus-cost-governing-it-on-a-reduced-budget.htm>
- [2] Agile Requirements Change Management, Retrieved 2010/01/04, <http://www.computer.org/portal/web/csdl/doi/10.1109/EURMIC.1998.708102>
- [3] Importance of Software Test Automation using tools such as QTP, Retrieved 2010-01-04, <http://qualitypoint.blogspot.com/2009/04/importance-of-software-test-automation.html>
- [4] Cost Benefits Analysis of Test Automation, Retrieved 2010/01/04, <http://www.softwarequalitymethods.com/Papers/Star99%20model%20Paper.pdf>

External links

- nFocus Testing as a Service (<http://www.nfocus.co.uk/Services/ManagedServices/TestingasaService.aspx>)
- Initto Testing as a Service (<http://www.initto.com/test-service.html>)
- Qutesys Testing as a Service (<http://www.qutesys.com/2011/01/testing-as-service-taas.html>)
- Manual Intervention in an Automated Test (<http://blog.nfocus.co.uk/2009/11/manual-intervention-in-automated-test.html>)
- Using Non-Functional Tests Tools with Axe and WatiN (<http://blog.nfocus.co.uk/2009/10/using-non-functional-tests-tools-with.html>)
- Practical Experience in Automated Testing (<http://www.methodsandtools.com/archive/archive.php?id=33>)
- Test Automation: Delivering Business Value (http://www.applabs.com/internal/app_whitepaper_test_automation_delivering_business_value_1v00.pdf)
- Guidelines for Test Automation framework (http://info.allianceglobalservices.com/Portals/30827/docs/test_automation_framework_and_guidelines.pdf)
- Tieto Testing as a Service (<http://www.tieto.com/default.asp?path=1,127,41042>)
- TestLab² Testing as a Service (<http://www.testlab2.com/>)
- Using Cloud Computing to Automate Full-Scale System Tests (<http://www.youtube.com/watch?v=atyq-41Gnjc>)
- The Cloud's Next Big Thing: Software Testing (http://www.informationweek.com/cloud-computing/blog/archives/2009/06/the_clouds_next.html)

Testing Maturity Model

The **Testing Maturity Model (TMM)** was based on the Capability Maturity Model, and first produced by the Illinois Institute of Technology.^{[1] [2]}

Its aim to be used in a similar way to CMM, that is to provide a framework for assessing the maturity of the test processes in an organisation, and so providing targets on improving maturity.

There are five levels of maturity as follows:

Level 1 - Initial At this level an organisation is using ad-hoc methods for testing, so results are not repeatable and there is no quality standard.

Level 2 - Definition At this level testing is defined a process, so there might be test strategies, test plans, test cases, based on requirements. Testing does not start until products are completed, so the aim of testing is to compare products against requirements.

Level 3 - Integration At this level testing is integrated into a software life cycle, e.g. the V-model. The need for testing is based on risk management, and the testing is carried out with some independence from the development area.

Level 4 - Management and Measurement At this level testing activities take place at all stages of the life cycle, including reviews of requirements and designs. Quality criteria are agreed for all products of an organisation (internal and external).

Level 5 - Optimisation At this level the testing process itself is tested and improved at each iteration. This is typically achieved with tool support, and also introduces aims such as defect prevention through the life cycle, rather than defect detection (zero defects).

Each level from 2 upwards has a defined set of processes and goals, which lead to practices and sub-practices.

The TMM has been since replaced^[3] by the Test Maturity Model integration and is now managed by the TMMI Foundation.^[4]

References

- [1] Article in Crosstalk (<http://www.crosstalkonline.org/storage/issue-archives/1998/199811/199811-Burnstein.pdf>), I. Burnstein, A. Homyen, R. Grom and C.R. Carlson, "A Model to Assess Testing Process Maturity", CROSSTALK 1998, Software Technology Support Center, Hill Air Force Base, Utah
- [2] Article on CM Crossroads (<http://www.cmcrossroads.com/component/content/6948?task=view>), I. Burnstein, L. Miller, "Testing Maturity Model (TMM) Certification", CM Crossroads 2002
- [3] TMMi reference (<http://www.tmmifoundation.org/html/tmmiref.html>), Sources of TMMi
- [4] TMMi Foundation (<http://www.tmmifoundation.org/>), TMMi Foundation

The article describing this concept was first published in: Crosstalk, August and September 1996 "Developing a Testing Maturity Model: Parts I and II", Ilene Burnstein, Taratip Suwannasart, and C.R. Carlson, Illinois Institute of Technology (article not in online archives at Crosstalk online (<http://www.crosstalkonline.org/back-issues>) anymore)

Testware

Generally speaking, **Testware** is a sub-set of software with a special purpose, that is, for software testing, especially for software testing automation. Automation testware for example is designed to be executed on automation frameworks.

Testware is an umbrella term for all utilities and application software that serve in combination for testing a software package but not necessarily contribute to operational purposes. As such, testware is not a standing configuration but merely a working environment for application software or subsets thereof.

It includes artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing. ^[1]

Testware is produced by both verification and validation testing methods. Like software, Testware includes codes and binaries as well as test cases, test plan, test report and etc. Testware should be placed under the control of a configuration management system, saved and faithfully maintained.

Compared to general software, testware is special because it has:

1. a different purpose
2. different metrics for quality and
3. different users

The different methods should be adopted when you develop testware with what you use to develop general software.

Testware is also referred as test tools in a narrow sense. ^[2]

References

- [1] Fewster, M.; Graham, D. (1999), *Software Test Automation, Effective use of test execution tools*, Addison-Wesley, ISBN 0-201-33140-3
- [2] http://www.homeoftester.com/articles/what_is_testware.htm

Think aloud protocol

Think-aloud protocol (or think-aloud protocols, or TAP) is a method used to gather data in usability testing in product design and development, in psychology and a range of social sciences (e.g., reading, writing and translation process research). The think-aloud method was introduced in the usability field by Clayton Lewis^[1] while he was at IBM, and is explained in *Task-Centered User Interface Design: A Practical Introduction* by C. Lewis and J. Rieman.^[2] The method was developed based on the techniques of protocol analysis by Ericsson and Simon.^{[3] [4] [5]}

Think aloud protocols involve participants thinking aloud as they are performing a set of specified tasks. Users are asked to say whatever they are looking at, thinking, doing, and feeling, as they go about their task. This enables observers to see first-hand the process of task completion (rather than only its final product). Observers at such a test are asked to objectively take notes of everything that users say, without attempting to interpret their actions and words. Test sessions are often audio and video taped so that developers can go back and refer to what participants did, and how they reacted. The purpose of this method is to make explicit what is implicitly present in subjects who are able to perform a specific task.

A related but slightly different data-gathering method is the **talk-aloud protocol**. This involves participants only describing their action but not giving explanations. This method is thought to be more objective in that participants merely report how they go about completing a task rather than interpreting or justifying their actions (see the standard works by Ericsson & Simon).

As Hannu and Pallab^[6] state the thinking aloud protocol can be divide in two different experimental procedures: the first one, is the concurrent thinking aloud protocol, collected during the decision task; the second procedure is the retrospective thinking aloud protocol gathered after the decision task.

References

- [1] Lewis, C. H. (1982). Using the "Thinking Aloud" Method In Cognitive Interface Design. Technical Report IBM RC-9265.
- [2] http://grouplab.cpsc.ucalgary.ca/saul/hci_topics/tcsd-book/chap-1_v-1.html *Task-Centered User Interface Design: A Practical Introduction*, by Clayton Lewis and John Rieman.
- [3] Ericsson, K., & Simon, H. (May 1980). "Verbal reports as data". *Psychological Review* **87** (3): 215–251. doi:10.1037/0033-295X.87.3.215.
- [4] Ericsson, K., & Simon, H. (1987). "Verbal reports on thinking". In C. Faerch & G. Kasper (eds.). *Introspection in Second Language Research*. Clevedon, Avon: Multilingual Matters. pp. 24–54.
- [5] Ericsson, K., & Simon, H. (1993). *Protocol Analysis: Verbal Reports as Data* (2nd ed.). Boston: MIT Press. ISBN 0262050293.
- [6] Hannu, K., & Pallab, P. (2000). "A comparison of concurrent and retrospective verbal protocol analysis". *American Journal of Psychology* (University of Illinois Press) **113** (3): 387–404. doi:10.2307/1423365. JSTOR 1423365. PMID 10997234.

Tiger team

A **tiger team** is a group of experts assigned to investigate and/or solve technical or systemic problems. The term may have originated in aerospace design but is also used in other settings, including information technology and emergency management. According to a 1964 definition, "In case the term 'tiger team' is unfamiliar to you, it has been described as 'a team of undomesticated and uninhibited technical specialists, selected for their experience, energy, and imagination, and assigned to track down relentlessly every possible source of failure in a spacecraft subsystem.'"^[1]

Security tiger teams

In security work, a **tiger team** is a specialized group that tests an organization's ability to protect its assets by attempting to circumvent, defeat, or otherwise thwart that organization's internal and external security. The term originated within the military to describe a team whose purpose is to penetrate security of "friendly" installations to test security measures. It now more generally refers to any team that attacks a problem aggressively.

Examples of tiger teams

1. The NSA Cyber Defense Tiger Team (Red Cell) is a tiger team that was created by the National Security Agency^[2]
2. Tiger Team, BAE Systems, Portsmouth Naval Dockyard. International Fast Response Ship Repair team
3. Many tiger teams are informally constituted through managerial edicts. One of these was set up in NASA circa 1966 to solve the "Apollo Navigation Problem" and it makes an interesting story. The motivation was the discovery that current technology was unable to navigate Apollo at the level of precision mandated by the mission planners. Tests using radio tracking data from unmanned Lunar Orbiter spacecraft to evaluate circumlunar Apollo navigation were revealing errors of 2000 meters instead of the 200 that the mission required to safely land Apollo when descending from its lunar orbit. For example, Apollo astronauts were practicing landings in safe areas using the simulators at Houston. A tenfold increase in this error-bound implied a hundredfold increase in the target area, which then included unacceptably dangerous terrain. The mission was seriously at risk. This was a navigation problem and so five tiger teams were set up to find and correct the problem, one at each NASA center, from CalTech JPL in the west to Goddard SFC (GSFC) in the east. The Russians via Luna 10 were also well aware of this problem. There was an intentionally competitive aspect to this strategy, which was "won" by JPL in the spring of 1968 when it was shown that the problem was caused by the unexpectedly large local gravity anomalies on the moon arising from large ringed maria, mountain ranges and craters on the moon. This also led to the construction of the first detailed gravimetric map of a body other than the earth and the discovery of the lunar mass concentrations (Mascons).^[3]


References

- [1] J. R. Dempsey, W. A. Davis, A. S. Crossfield, and Walter C. Williams, "Program Management in Design and Development," in Third Annual Aerospace Reliability and Maintainability Conference, Society of Automotive Engineers, 1964, p. 7–8 ([http://books.google.com/books?id=lv0pAQAAIAAJ&q="tiger team"](http://books.google.com/books?id=lv0pAQAAIAAJ&q=)).
- [2] NSA.gov (http://www.nsa.gov/public_info/speeches_testimonies/nsa_videos/cyber_defense_2008.shtml)
- [3] Paul Muller and William Sjogren (1968). "Mascons: lunar mass concentrations". *Science* 161 (3842): 680–684. doi:10.1126/science.161.3842.680. PMID 17801458

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

Tosca (Software)

TOSCA Testsuite

	
Original author(s)	TRICENTIS Technology & Consulting
Developer(s)	TRICENTIS Technology & Consulting
Stable release	7.3.0 / June 20, 2011
Written in	C#, Java, VB6
Operating system	Microsoft Windows
Available in	English, German
Type	Test automation
License	Trial version available
Website	tricentis.com ^[1]

TOSCA Testsuite is a software tool for the automated execution of functional and regression software testing. In addition to test automation functions, TOSCA includes integrated test management, a graphical user interface (GUI), a command line interface (CLI) and an application programming interface (API). TOSCA Testsuite is developed by the Austrian software company TRICENTIS Technology & Consulting GmbH based in Vienna. In 2011, TOSCA was included in Gartner Inc.'s "Magic Quadrant for Integrated Software Quality Suites" report as a "visionary".^[2]

Architecture

TOSCA is a test management, design, execution and data generation toolset for functional and regression tests.^[3] TOSCA Testsuite consists of the following:

- TOSCA Commander, the testsuite's execution tool, is used to create, administer, execute and analyze test cases.^[4]
- TOSCA Wizard, builds a model of the application by storing the technical information XML-GUI Maps called modules.^[5]
- Once test cases have been created, TOSCA Executor, executes the test cases and displays the results in TOSCA Commander.
- TOSCA Exchange Portal, a portal where customers can use and exchange special modules, extensions and prebuilt TOSCA Commander components (subsets).
- The Test Repository, which includes integrated version control, stores all test assets and can be accessed by multiple users.

Functionality

Business dynamic steering: the concept behind TOSCA Commander is a model-driven approach to make "the entire test, and not just the input data, dynamic".^[6] Test cases are built by dragging and dropping modules and entering validation values and actions.^[7] The dynamization of the test is supposed to enable a business-based description of manual and automated test cases so test cases can be designed, specified, automated and maintained by non-technical users (SMEs).^[8]

The main features of Tosca Testsuite include the generation of dynamic, synthetic test data, highly automated business dynamic steering of test case generation and the unified handling and executing of manual and automated as well as GUI and non-GUI tests.^[9]

In addition, test cases can be weighted according to their importance in the smooth running of the business process. In this way, TOSCA provides detailed reporting, which shows the impact of existing technical weak points on the fulfillment of requirements. Fecher, for example, uses the test tool in new developments and application and database migration projects.^[10]

The following two limitations have been identified in comparison with other test automation solutions:

- No load or stress testing^[11]
- No Active-X components

Extensions

In addition to the basic software, there are the following extensions:

- Requirements: requirements are imported, exported, edited and administrated. The requirements are risk weighted and then linked to the test cases after test case design.
- TestCase-Design Workbench: defines, on the basis of the requirements, which test cases are needed to cover the specific test object and then generates test cases, employing all combinations: pairwise, orthogonal array and linear expansion.^{[12] [13]}
- Reporting: test results are collected, analyzed and presented. Reports can be created using Crystal Reports or exported as a PDF or XML file.^[14]
- TOSCA Easy Entrance: creates reusable entities through drag and drop.
- User management: multi-user concept with integrated check-in and check-out mechanisms and versioning.
- WebAccess: TOSCA Testsuite provides remote access in real time through WebAccess.
- PDF comparison and bidirectional communication with Microsoft Word and Microsoft Excel.

Supported technologies

The automation of software tests is supported for the following technologies:

- Programming languages and frameworks: Delphi, .NET including WPF, Java Swing/SWT/AWT, Visual Basic
 - Application development environments: Gupta, PowerBuilder
 - Web browsers: Internet Explorer, Firefox
 - Host applications in 3270, 5250
 - Key application programs: SAP, Siebel
 - Single-position application programs: Microsoft Outlook, Microsoft Excel
 - Hardware & protocols: USB execution, Flash, SOAP (WebServices), ODBC
-

System environment

TOSCA Testsuite supports the following operating systems:

- Windows XP, SP 2 and above
- Windows Vista SP 2
- Windows 7 (both 32-bit and 64-bit versions)^[15]

The following databases are supported for multi-user operation:

- Microsoft®SQL Server 2005
- Oracle 10g
- DB2 v.9.1.

User, industry and best practice solutions

As of November 2008, 140 customers were using Tosca, 70 per cent of them in Germany. This includes the German Stock Exchange, where TOSCA is in continuous test operation. In Austria, the program is in use in numerous banks, insurance, telecommunication and industrial companies such as OMV or EVN (Energieversorgung Niederösterreich or Lower Austrian Energy Supply).^[16] TOSCA is one of the test tools covered in the Business Process Management 2 course of the IT & Business Informatics program of Campus02.^[17] There are the following industry and best practice solutions:

- TOSCA@SAP is a best practice solution for using TOSCA Testsuite in SAP environments.
- TOSCA@data is a best practice solution for test case design and the automated generation of synthetic test data with TOSCA Testsuite. In contrast to other solutions, real production data is not used.^{[18] [19]}
- TOSCA@energy is a best practice solution for energy suppliers, which uses TOSCA Testsuite to comply with European Union and national regulations.

References in books and periodicals

- Die TOSCA-Testsuite von TRICENTIS. In: Harry M. Sneed, Manfred Baumgartner, Richard Seidl: Der Systemtest: Von den Anforderungen zum Qualitätsnachweis, Hanser, München 2009, ISBN 978-3-446-41708-3, p. 224–229.
- Edward Bishop: Changing tests weakens them^[20]. In: Professional Tester, September 2010, ISSN 1742-8742, S. 13–15. (Bishop was the editor-in-chief of the journal at the time of publication.)

External links

- Official web page of TOSCA Testsuite^[21]

References

- [1] <http://tricentis.com/en>
- [2] <http://www.gartner.com/technology/media-products/reprints/microfocus/vol4/article1/article1.html>
- [3] <http://www.professionaltester.com/magazine/backissue/5/ProfessionalTesterNovember2010-Bishop.pdf>
- [4] Harry M. Sneed, Manfred Baumgartner, Richard Seidl, Der Systemtest: Von den Anforderungen zum Qualitätsnachweis, (München: Carl Hanser Verlag München, 2009), 226
- [5] <http://www.iceteagroup.com/LinkClick.aspx?fileticket=ixGsyqQuv0g%3D&tabid=272>
- [6] <http://www.iceteagroup.com/LinkClick.aspx?fileticket=ixGsyqQuv0g%3D&tabid=272>
- [7] <http://www.professionaltester.com/magazine/backissue/5/ProfessionalTesterNovember2010-Bishop.pdf>
- [8] Harry M. Sneed, Manfred Baumgartner, Richard Seidl, Der Systemtest: Von den Anforderungen zum Qualitätsnachweis, (München: Carl Hanser Verlag München, 2009), 225
- [9] <http://www.it-media.at/article.php?articleid=2973&backbuttonurl=%2Fittbusiness-section.php%3Fsectionid%3D46>
- [10] http://www.innovations-report.de/html/berichte/cebit_2008/bericht-104677.html

- [11] <http://www.xqual.com/qa/tools.html>
- [12] <http://www.professionaltester.com/magazine/backissue/5/ProfessionalTesterNovember2010-Bishop.pdf>
- [13] Harry M. Sneed, Manfred Baumgartner, Richard Seidl, *Der Systemtest: Von den Anforderungen zum Qualitätsnachweis*, (München: Carl Hanser Verlag München, 2009), 225
- [14] <http://www.computerwelt.at/detailArticle.asp?a=115726&n=2>
- [15] <http://www.microsoft.com/windows/compatibility/Windows-7/en-us/Details.aspx?type=Software&p=TOSCA%20TestSuite&v=Tricentis%20Technology%20%26%20Consulting&uid=7&pf=5&pi=8&c=Development%20Tools&sc=all&os=64-bit>
- [16] <http://derstandard.at/1226067135107>
- [17] http://itmkb.campus02.at/index.php/Business_Process_Management_2_%28ITMAS_3._Sem%29#GP_Knowledgebase
- [18] <http://www.it-media.at/article.php?articleid=2973&backbuttonurl=%2Fittbusiness-section.php%3Fsectionid%3D46>
- [19] <http://www.wirtschaftsblatt.at/home/schwerpunkt/itnews/TechNews/426206/index.do>
- [20] <http://www.professionaltester.com/magazine/backissue/5/ProfessionalTesterNovember2010-Bishop.pdf>
- [21] <http://www.tricentis.com/en>

TPS report

A **TPS report** (Testing Procedure Specification) is a document used in software engineering, in particular by a Software Quality Assurance group or individual, that describes the testing procedures and the testing process.

The official definition and creation is provided by the Institute of Electrical and Electronics Engineers (IEEE) as follows:

IEEE 829 - Test Procedure Specification

The Test Procedures are developed from both the Test Design and the Test Case Specification. The document describes how the tester will physically run the test, the physical set-up required, and the procedure steps that need to be followed. The standard defines ten procedure steps that may be applied when running a test.

Popular culture references

Office Space

After its use in the comedic film *Office Space*, "TPS report" has come to connote pointless mindless paperwork,^[1] and an example of "literacy practices" in the work environment that are "meaningless exercises imposed upon employees by an inept and uncaring management" and "relentlessly mundane and enervating".^[2] According to the film's writer and director Mike Judge, the acronym stood for "Test Program Set" in the movie.^[3] In the story, the protagonist is reprimanded by several of his superiors for forgetting to put the new cover sheet on his TPS report. When one of the efficiency consultants that visits the firm asked "What does TPS stand for?", the audio cut away so that the term was left undefined in the released film.

Notes


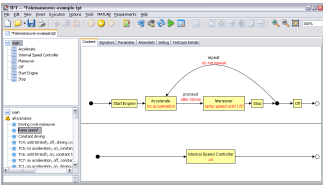
- [1] Steven S. Little, *The Milkshake Moment: Overcoming Stupid Systems, Pointless Policies and Muddled Management to Realize Real Growth* (John Wiley & Sons, 2008), ISBN 9780470257463, p.51. Excerpt available (<http://books.google.com/books?id=JodaaDKkd0YC&pg=PA51>) at Google Books.
- [2] Bronwyn T. Williams, Amy A. Zenger, *Popular Culture and Representations of Literacy* (Routledge/Taylor & Francis, 2007), ISBN 978-0415360951, p. 61. Excerpt available (<http://books.google.com/books?id=IQQJoEWyLD8C&pg=PA61>) at Google Books.
- [3] <http://www.rollingstone.com/music/news/15765/92346>

External links

- IEEE 829-1998 - Standard for Software Test Documentation (http://standards.ieee.org/reading/ieee/std_public/description/se/829-1998_desc.html)

TPT (Software)

Time Partition Testing (TPT)

	
	
Developer(s)	PikeTec GmbH ^[1]
Stable release	3.4.2 / August 2011
Operating system	Cross-platform, Windows
License	Proprietary
Website	TPT product page ^[2]

TPT (*Time Partition Testing*) is a systematic test methodology for the automated software test and verification of embedded control systems. TPT is specialized on testing and validation embedded systems whose inputs and outputs can be represented as signals and is a dedicated method for continuous behavior of systems.^[3] Most control systems belong to this system class. The outstanding characteristic of control systems is the fact that they interact closely interlinked with a real world environment. Controller need to observe their environment and react correspondingly to its behaviour^[4]. The system works in an interactional cycle with its environment and is subject to temporal constraints. Testing these systems is to stimulate and to check the timing behavior. Traditional functional testing methods use scripts - TPT uses model-based testing.

TPT combines a systematic and graphic modelling technique for test cases with a fully automatic test execution in different environments and automatic test evaluation. TPT covers the following four test activities:

- test case modeling,
- test execution in different environments (automated)
- test result analysis (test assessment (automated))
- test documentation (automated)

Graphic Test Cases

The exact process of individual test cases is modelled graphically with the aid of special state machines and time partitioning^[5] in TPT. Natural language texts as an element of the graphics support the simple and demonstrative readability even for non-programmers. Substantial techniques such as parallel and hierarchical state machines, conditional branching, signal description as well as measured signals allow an intuitive and graphic modelling even of complex test cases.

Systematic Test Cases

TPT was developed specifically for testing of continuous and reactive behaviour of embedded systems. TPT can be seen as the extension of the classification tree method in terms of timing behavior. Because of its systematic approach in test case generation, TPT even keeps track of very complex systems whose thorough testing requires a large amount of test cases thus making it possible to find failures in the system under test with an ideal amount of test cases.

The underlying idea of TPT's systematic is the separation of similarities and differences among the test cases: most test cases are very similar in their structural process and can "only" be differentiated in a few, but crucial details^[6]. TPT makes use of this fact by jointly modelling and using joint structures. On the one hand, redundancies are thus avoided. On the other hand, it is made very clear what the test cases actually differ in – i.e. which specific aspect they respectively test. The comparability of test cases and thus the overview is improved in this approach and the attention of the tester is focused on the essential – the differentiating features of the test cases.

The hierarchical structure of the test cases makes it possible to break complex test problems down into sub-problems thus also improving the clarity and – as a result – the quality of the test.

These modelling techniques support the tester in finding the actually relevant cases, avoiding redundancies and keeping track of even large numbers of test cases.^[7]

Reactive Tests

With TPT, each test case can specifically react to the system's behaviour during the testing process in real time – for instance in order to react on the system exactly when a certain system-state occurs or a sensor signal exceeds a certain threshold. If, for example, a sensor failure for an engine controller is to be simulated when the engine idling speed is exceeded, it has to be possible to react to the event "engine idling speed exceeded" in the description of the test case.

Test Execution

TPT test cases are made independent of its execution. The test cases can be executed in real time in almost any environment due to the so called virtual machine (VM) concept. Examples are MATLAB/Simulink, TargetLink, ASCET, C-Code, CAN, LIN, AUTOSAR, LABCAR, INCA, Software-in-the-Loop (SiL) and HiL. Thus TPT is an integrated tool to be used in all testing phases of the development like unit testing, integration testing, system testing and regression testing.

TPT virtual machine

The modeled test cases in TPT are compiled and during test execution interpreted by the so called virtual machine (VM). The VM is the same for all platforms and all tests. Only a platform adapter realizes the signal mapping for the individual application. The TPT-VM is implemented in ANSI C and requires a memory of just a few kilobytes and can completely do without a dynamic memory allocation, allowing it to be applied in minimalist and environments with few resources too. TPT's Virtual Machine is able to process tests in real time with defined response behaviour. The response times of TPT test cases are normally given within micro seconds – depending on the complexity and test hardware.

Programmed Test Assessment

The expected system behaviour for individual test cases should also be automatically tested in order to assure efficient test processes. TPT offers the possibility to compute the properties for the expected behaviour online (during test execution) and offline (after test execution). While online evaluation uses the same modelling techniques as test modelling, offline evaluation offers decidedly more far-reaching possibilities for more complex evaluations, including operations such as comparisons with external reference data, limit-value monitoring, signal filters, analyses of state sequences and time conditions.

The offline evaluation is, technically speaking, based on the Python script language, which has been extended by specific syntactic language elements and a specialized evaluation library in order to give optimal support to the test evaluation. The use of a script language ensures a high degree of flexibility in the test evaluation: access to reference data, communication with other tools and development of one's own domain-specific libraries for test evaluation is supported. Besides of the script based test result evaluation user interfaces provide simple access to the test assessments and help non-programmers to avoid scripting.

Measurement data from other sources like TargetLink or Simulink signal logging or MCD-3 measurement data can be assessed automatically. This data can be independent from the test execution.

Test Documentation

TPT test documentation according to IEEE 829 presents the result of the test evaluation to the tester in a HTML, report, in which not only the pure information "success", "failed" or "unknown" can be depicted as the test result for each test case, but also details such as characteristic parameters or signals that have been observed in the test execution or computed in the test evaluation. Since the test assessment returns proper information about the timing and the checked behavior this information can be made available in the report. The content of the test documentation as well as the structure of the document can be freely configured with the help of a template.

Requirements Tracing

Industry norms such as IEC 61508, DO-178B, EN 50128 and ISO 26262 require traceability of requirements and tests. TPT offers an interface to requirements tools like Telelogic DOORS in order to support these activities.

Application

TPT is a model-based testing tool and is applied mainly in the automotive controller development and has originally been developed within Daimler AG for their own development. The first release of the test tool has been used in 2000. Daimler coordinated the development of the tool for years. Now PikeTec GmbH continues the development of the tool and is used by many different other car manufacturers and suppliers as Robert Bosch GmbH, Continental and Hella.

External links

- PikeTec ^[8]

References

- [1] <http://www.piketec.com>
- [2] <http://www.piketec.com/products/tpt.php?lang=en>
- [3] Justyna Zander-Nowicka, Abel Marrero Pérez, Ina Schieferdecker, Zhen Ru Dai: Test Design Patterns for Embedded Systems, In: 10th International Conference on Quality Engineering in Software Technology, CONQUEST 2007, Potsdam, Germany, September 2007 (http://www.fokus.fraunhofer.de/de/motion/ueber_motion/unser_team/zander_justyna/Conquest_07_vfinal_DV2.pdf)
- [4] Karl J. Åström and Richard M. Murray (2008). *Feedback Systems: An Introduction for Scientists and Engineers*. (http://www.cds.caltech.edu/~murray/books/AM08/pdf/am08-complete_22Feb09.pdf). Princeton University Press. ISBN 0691135762. .
- [5] Schieferdecker, Bringmann, Grossmann: Continuous TTCN-3: Testing of Embedded Control Systems, In: Proceedings of 28th International Conference on Software Engineering, Shanghai, China, 2006 (<http://www.irisa.fr/lande/lande/icse-proceedings/seas/p29.pdf>)
- [6] Lehmann, TPT - Dissertation, 2003 (<http://www.piketec.com/downloads/papers/ELehmannDissertation.pdf>)
- [7] Lehmann: Time Partition Testing: A Method for Testing Dynamical Functional Behavior IN: Proceedings of Test2000, Lindon, Great Britain, 2000 (http://evotest.iti.upv.es/index.php?option=com_docman&task=doc_view&gid=29&Itemid=70)
- [8] <http://www.piketec.com/products/tpt.php>
- Bringmann, Krämer: Systematic testing of the continuous behavior of automotive systems In: International Conference on Software Engineering: Proceedings of the 2006 international workshop on Software, Shanghai, China, 2006 (<http://portal.acm.org/citation.cfm?id=1138479>)
- Springer Berlin / Heidelberg (<http://www.springerlink.com/content/j725838472n68171/>)
- Eckard Bringmann, Andreas Krämer. Model-Based Testing of Automotive Systems, In: ICST, pp.485-493, 2008 International Conference on Software Testing, Verification, and Validation, 2008. (http://www.piketec.com/downloads/papers/Kraemer2008-Model_based_testing_of_automotive_systems.pdf)
- Grossmann, Müller: A Formal Behavioral Semantics for TestML (<http://jerry.c-lab.de/~wolfgang/isola06.pdf>)
- Justyna Zander-Nowicka, Abel Marrero Pérez, Ina Schieferdecker, Zhen Ru Dai: Test Design Patterns for Embedded Systems, In: 10th International Conference on Quality Engineering in Software Technology, CONQUEST 2007, Potsdam, Germany, September 2007 (http://www.fokus.fraunhofer.de/de/motion/ueber_motion/unser_team/zander_justyna/Conquest_07_vfinal_DV2.pdf)

References

- [1] Carlos, Tom (2008-10-21). Requirements Traceability Matrix - RTM. PM Hut, 21 October 2008. Retrieved on 2009-10-17 from <http://www.pmhut.com/requirements-traceability-matrix-rtm>.

External links

- Bidirectional Requirements Traceability (<http://www.compaid.com/caiinternet/ezine/westfall-bidirectional.pdf>) by Linda Westfall
- Requirements Traceability (http://www.projectperfect.com.au/info_requirements_traceability.php) Neville Turbit
- Software Development Life Cycles: Outline for Developing a Traceability Matrix (<http://www.regulatory.com/forum/article/tracedoc.html>) by Diana Baldwin
- StickyMinds article: Traceability Matrix (http://www.stickyminds.com/r.asp?F=DART_6051) by Karthikeyan V
- Why Software Requirements Traceability Remains a Challenge (<http://www.crosstalkonline.org/storage/issue-archives/2009/200907/200907-Kannenber.pdf>) by Andrew Kannenberg and Dr. Hossein Saiedian

Tree testing

Tree testing is a usability technique for evaluating the findability of topics in a website. It is also known as **reverse card sorting** or **card-based classification**.^[1]

A large website is typically organized into a hierarchy (a "tree") of topics and subtopics. Tree testing provides a way to measure how well users can find items in this hierarchy.

Unlike traditional usability testing, tree testing is not done on the website itself; instead, a simplified text version of the site structure is used. This ensures that the structure is evaluated in isolation, nullifying the effects of navigational aids, visual design, and other factors.

Basic method

In a typical tree test:

1. The participant is given a "find it" task (e.g., "Look for brown belts under \$25").
 2. They are shown a text list of the top-level topics of the website.
 3. They choose a heading, and are then shown a list of subtopics.
 4. They continue choosing (moving down through the tree) — drilling down, backtracking if necessary — until they find a topic that satisfies the task (or until they give up).
 5. The participant does several tasks in this manner, starting each task back at the top of the tree.
 6. Once several participants have completed the test, the results are analyzed for each task.
-

Analyzing the results

The analysis typically tries to answer these questions:

- Could users successfully find particular items in the tree?
- Could they find those items directly, without having to backtrack?
- If they couldn't find items, where did they go astray?
- Could they choose between topics quickly, without having to think too much?
- Overall, which parts of the tree worked well, and which fell down?

Tools

Tree testing was originally done on paper (typically using index cards), but can now also be conducted using specialized software.

References

- [1] Donna Spencer (April 2003). "Card-Based Classification Evaluation" (http://www.bboxesandarrows.com/view/card_based_classification_evaluation). .

External links

- Treejack, tree-testing software by Optimal Workshop (<http://www.optimalworkshop.com/treejack.htm>)
- C-Inspector, tree-testing software by Steffen Schilb (<http://www.c-inspector.com/index.php>)
- Dave O'Brien (Dec 2009). Tree Testing: A quick way to evaluate your IA (<http://www.bboxesandarrows.com/view/tree-testing>)

TTCN-3

TTCN-3 (*Testing and Test Control Notation version 3*) is a strongly typed test scripting language used in conformance testing of communicating systems and a specification of test infrastructure interfaces that glue abstract test scripts with concrete communication environments. TTCN-3 has been developed by ETSI and its predecessor is TTCN-2. Despite sharing same fundamental concepts, TTCN-2 and TTCN-3 are essentially two different languages, the latter having simpler syntax and standardized adapter interfaces. TTCN-3 scripts can be combined with ASN.1 type definitions. ASN.1 is natively supported by major TTCN-3 tool vendors.

Applications

TTCN-3 has been used to deploy SIP, WiMAX, and DSRC test systems.

The Open Mobile Alliance has recently adopted a strategy of using TTCN-3 for translating some of the test cases in an enabler test specification into an executable representation.^[1]

The AUTOSAR project is promoting the use of TTCN-3 within the automotive industry.^[2]

Architecture

A typical TTCN-3 test system consists of:

- execution core that runs test cases (TE or test execution)
 - SUT adapter implementing TRI SA interface that is responsible for network interface code
 - platform adapter implementing TRI PA interface that is responsible for timers and external functions
 - coding and decoding (TCI-CD interface)
-

- test control interface that uses TCI-TM API

References

- [1] TTCN-3 Test Code Developments - Request for Information (http://www.openmobilealliance.org/documents/OMA-BOD-IOP-2008-0007R04-INP_TTCN_3_Test_Code_Developments_RFI.pdf), May 1, 2008, Open Mobile Alliance, retrieved on May 7, 2008
- [2] TTCN-3 application areas (<http://www.ttcn-3.org/ApplicationAreas.htm>), ETSI official TTCN-3 web site, retrieved on May 7, 2008

External links

- ETSI TTCN-3 web site (<http://www.ttcn-3.org>)
- Official TTCN-3 standard (<http://www.ttcn-3.org/StandardSuite.htm>)
- First TTCN-3 Quick Reference Card (<http://www.blukactus.com/TTCN3-QRC.html>)
- TTCN-3 language reference (http://wiki.openttcn.com/media/index.php/OpenTTCN/Language_reference)
- BTT - BroadBit Test Tool, open-source, multi-platform TTCN-3 test tool (http://www.broadbit.net/portal/?page_id=392)
- Elvior TestCast Test Tool, commercial, full-feature TTCN-3 test tool (<http://elvior.com/testcast/introduction>)

Twist (software)

Twist

Initial release	October 6, 2008 ^[1]
Stable release	2.0 / March 1, 2010
Operating system	Cross-platform
Type	Test automation, Agile testing
License	Proprietary, free trial
Website	www.thoughtworks-studios.com/Twist ^[2]

Twist is a test automation and functional testing solution built by Thoughtworks Studios, the software division of ThoughtWorks. It uses Behavior Driven Development (BDD) and Test-driven development (TDD)^[3] for functional testing of the application.^[4] It is a part of the Adaptive ALM solution^[5] comprising of Twist for Agile testing by ThoughtWorks Studios, Mingle for Agile project management and Go for Agile release management

Features

Twist allows test specifications to be written in English or any UTF-8 supported language. Test implementation is done using Java or Groovy. Twist's IDE supports manual, automated and hybrid testing.^[6] Twist can be used with any Java based driver. It provides support for Selenium and Sahi for testing web-based applications, SWTBot for testing Eclipse/SWT applications and Frankenstein for testing Java Swing applications.

References

- [1] Sims, Chris (2008-10-06). "ThoughtWorks Announces Twist, Automated Functional Testing Platform" (<http://www.infoq.com/news/2008/10/twist-announced>). *InfoQ*. .
- [2] <http://www.thoughtworks-studios.com/agile-test-automation>
- [3] Rubinstein, David (2010-03-01). "ThoughtWorks puts new twist on testing" (<http://www.sdtimes.com/link/34154>). *SD Times*. .
- [4] "Agile Test Automation Tool Updated" (<http://www.drdoobs.com/tools/223100961>). *Dr. Dobb's Journal*. 2010-03-01. .
- [5] "Thoughtworks Studios Releases Adaptive ALM" (<http://www.drdoobs.com/tools/219400175>). *Dr. Dobb's Journal*. 2009-08-17. .
- [6] Penchikala, Srin (2010-04-05). "Twist 2.0 Supports Behavior Driven and Collaborative Testing" (<http://www.infoq.com/news/2010/04/twist-2.0>). *InfoQ*. .

External links

- Twist Agile Testing (<http://www.thoughtworks-studios.com/agile-test-automation>)
- Twist Community (<http://community.thoughtworks.com/groups/c63fc04bc9/summary>)

Unit testing

In computer programming, **unit testing** is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. In object-oriented programming a unit is usually an interface, such as a class. Unit tests are created by programmers or occasionally by white box testers during the development process.

Ideally, each test case is independent from the others: substitutes like method stubs, mock objects,^[1] fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.

Benefits

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct.^[2] A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development cycle.

Facilitates change

Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.

Readily-available unit tests make it easy for the programmer to check whether a piece of code is still working properly.

In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained.

Simplifies integration

Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.

An elaborate hierarchy of unit tests does not equal integration testing. Integration with peripheral units should be included in integration tests, but not in unit tests. Integration testing typically still relies heavily on humans testing manually; high-level or global-scope testing can be difficult to automate, such that manual testing often appears faster and cheaper.

Documentation

Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's API.

Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

By contrast, ordinary narrative documentation is more susceptible to drifting from the implementation of the program and will thus become outdated (e.g., design changes, feature creep, relaxed practices in keeping documents up-to-date).

Design

When software is developed using a test-driven approach, the unit test may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behaviour. The following Java example will help illustrate this point.

Here is a test class that specifies a number of elements of the implementation. First, that there must be an interface called `Adder`, and an implementing class with a zero-argument constructor called `AdderImpl`. It goes on to assert that the `Adder` interface should have a method called `add`, with two integer parameters, which returns another integer. It also specifies the behaviour of this method for a small range of values.

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        assert (adder.add(1, 1) == 2);
        assert (adder.add(1, 2) == 3);
        assert (adder.add(2, 2) == 4);
        assert (adder.add(0, 0) == 0);
        assert (adder.add(-1, -2) == -3);
        assert (adder.add(-1, 1) == 0);
        assert (adder.add(1234, 988) == 2222);
    }
}
```

In this case the unit test, having been written first, acts as a design document specifying the form and behaviour of a desired solution, but not the implementation details, which are left for the programmer. Following the "do the simplest thing that could possibly work" practice, the easiest solution that will make the test pass is shown below.

```
interface Adder {
    int add(int a, int b);
}
class AdderImpl implements Adder {
    int add(int a, int b) {
        return a + b;
    }
}
```

Unlike other diagram-based design methods, using a unit-test as a design has one significant advantage. The design document (the unit-test itself) can be used to verify that the implementation adheres to the design. With the unit-test design method, the tests will never pass if the developer does not implement the solution according to the design.

It is true that unit testing lacks some of the accessibility of a diagram, but UML diagrams are now easily generated for most modern languages by free tools (usually available as extensions to IDEs). Free tools, like those based on the xUnit framework, outsource to another system the graphical rendering of a view for human consumption.

Separation of interface from implementation

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests

into integration tests, and when test cases fail, makes it less clear which component is causing the failure. See also Fakes, mocks and integration tests

Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher quality unit that is also more maintainable.

Parameterized Unit Testing (PUT)

Parameterized Unit Tests (PUTs) are tests that take parameters. Unlike traditional unit tests, which are usually closed methods, PUTs take any set of parameters. PUTs have been supported by JUnit 4 and various .NET test frameworks. Suitable parameters for the unit tests may be supplied manually or in some cases are automatically generated by the test framework. Various industrial testing tools also exist to generate test inputs for PUTs.

Unit testing limitations

Testing cannot be expected to catch every error in the program: it is impossible to evaluate every execution path in all but the most trivial programs. The same is true for unit testing. Additionally, unit testing by definition only tests the functionality of the units themselves. Therefore, it will not catch integration errors or broader system-level errors (such as functions performed across multiple units, or non-functional test areas such as performance). Unit testing should be done in conjunction with other software testing activities. Like all forms of software testing, unit tests can only show the presence of errors; they cannot show the absence of errors.

Software testing is a combinatorial problem. For example, every boolean decision statement requires at least two tests: one with an outcome of "true" and one with an outcome of "false". As a result, for every line of code written, programmers often need 3 to 5 lines of test code.^[3] This obviously takes time and its investment may not be worth the effort. There are also many problems that cannot easily be tested at all – for example those that are nondeterministic or involve multiple threads. In addition, writing code for a unit test is as likely to be at least as buggy as the code it is testing. Fred Brooks in *The Mythical Man-Month* quotes: *never take two chronometers to sea. Always take one or three.* Meaning, if two chronometers contradict, how do you know which one is correct?

To obtain the intended benefits from unit testing, rigorous discipline is needed throughout the software development process. It is essential to keep careful records not only of the tests that have been performed, but also of all changes that have been made to the source code of this or any other unit in the software. Use of a version control system is essential. If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.

It is also essential to implement a sustainable process for ensuring that test case failures are reviewed daily and addressed immediately.^[4] If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite, increasing false positives and reducing the effectiveness of the test suite.

Applications

Extreme Programming

Unit testing is the cornerstone of Extreme Programming, which relies on an automated unit testing framework. This automated unit testing framework can be either third party, e.g., xUnit, or created within the development group.

Extreme Programming uses the creation of unit tests for test-driven development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement isn't implemented yet, or because it intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, pass.

Most code in a system is unit tested, but not necessarily all paths through the code. Extreme Programming mandates a "test everything that can possibly break" strategy, over the traditional "test every execution path" method. This leads developers to develop fewer tests than classical methods, but this isn't really a problem, more a restatement of fact, as classical methods have rarely ever been followed methodically enough for all execution paths to have been thoroughly tested. Extreme Programming simply recognizes that testing is rarely exhaustive (because it is often too expensive and time-consuming to be economically viable) and provides guidance on how to effectively focus limited resources.

Crucially, the test code is considered a first class project artifact in that it is maintained at the same quality as the implementation code, with all duplication removed. Developers release unit testing code to the code repository in conjunction with the code it tests. Extreme Programming's thorough unit testing allows the benefits mentioned above, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. These unit tests are also constantly run as a form of regression test.

Unit testing is also critical to the concept of Emergent Design. As Emergent Design is heavily dependent upon Refactoring, unit tests are integral component.^[5]

Techniques

Unit testing is commonly automated, but may still be performed manually. The IEEE does not favor one over the other.^[6] A manual approach to unit testing may employ a step-by-step instructional document. Nevertheless, the objective in unit testing is to isolate a unit and validate its correctness. Automation is efficient for achieving this, and enables the many benefits listed in this article. Conversely, if not planned carefully, a careless manual unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing.

To fully realize the effect of isolation while using an automated approach, the unit or code body under test is executed within a framework outside of its natural environment. In other words, it is executed outside of the product or calling context for which it was originally created. Testing in such an isolated manner reveals unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated.

Using an automation framework, the developer codes criteria into the test to verify the unit's correctness. During test case execution, the framework logs tests that fail any criterion. Many frameworks will also automatically flag these failed test cases and report them in a summary. Depending upon the severity of a failure, the framework may halt subsequent testing.

As a consequence, unit testing is traditionally a motivator for programmers to create decoupled and cohesive code bodies. This practice promotes healthy habits in software development. Design patterns, unit testing, and refactoring often work together so that the best solution may emerge.

Unit testing frameworks

Unit testing frameworks are most often third-party products that are not distributed as part of the compiler suite. They help simplify the process of unit testing, having been developed for a wide variety of languages. Examples of testing frameworks include open source solutions such as the various code-driven testing frameworks known collectively as xUnit, and proprietary/commercial solutions such as TBrun, Testwell CTA++ and VectorCAST/C++.

It is generally possible to perform unit testing without the support of a specific framework by writing client code that exercises the units under test and uses assertions, exception handling, or other control flow mechanisms to signal failure. Unit testing without a framework is valuable in that there is a barrier to entry for the adoption of unit testing; having scant unit tests is hardly better than having none at all, whereas once a framework is in place, adding unit tests becomes relatively easy.^[7] In some frameworks many advanced unit test features are missing or must be hand-coded.

Language-level unit testing support

Some programming languages directly support unit testing. Their grammar allows the direct declaration of unit tests without importing a library (whether third party or standard). Additionally, the boolean conditions of the unit tests can be expressed in the same syntax as boolean expressions used in non-unit test code, such as what is used for `<syntaxhighlight lang="java" enclose="none"> if </syntaxhighlight> and <syntaxhighlight lang="java" enclose="none"> while </syntaxhighlight> statements.`

Languages that directly support unit testing include:

- Cobra
- D
- Java

Notes

[1] Fowler, Martin (2007-01-02). "Mocks aren't Stubs" (<http://martinfowler.com/articles/mocksArentStubs.html>). . Retrieved 2008-04-01.

[2] Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management* (<http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html>). Wiley-IEEE Computer Society Press. p. 75. ISBN 0470042125. .

[3] Cramblitt, Bob (2007-09-20). "Alberto Savoia sings the praises of software testing" (http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1273161,00.html). . Retrieved 2007-11-29.

[4] daVeiga, Nada (2008-02-06). "Change Code Without Fear: Utilize a regression safety net" (<http://www.ddj.com/development-tools/206105233>). . Retrieved 2008-02-08.

[5] http://www.agilesherpa.org/agile_coach/engineering_practices/emergent_design/

[6] IEEE Standards Board, "IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987" (http://iteso.mx/~pgutierrez/calidad/Estandares/IEEE_1008.pdf) in *IEEE Standards: Software Engineering, Volume Two: Process Standards; 1999 Edition; published by The Institute of Electrical and Electronics Engineers, Inc.* Software Engineering Technical Committee of the IEEE Computer Society.

[7] Bullseye Testing Technology (2006–2008). "Intermediate Coverage Goals" (<http://www.bullseye.com/coverage.html#intermediate>). . Retrieved 24 March 2009.

External links

- Unit Testing Guidelines from GeoSoft (<http://geosoft.no/development/unittesting.html>)
- Test Driven Development (Ward Cunningham's Wiki) (<http://c2.com/cgi/wiki?TestDrivenDevelopment>)
- Unit Testing 101 for the Non-Programmer (http://www.saravanansubramanian.com/Saravanan/Articles_On_Software/Entries/2010/1/19_Unit_Testing_101_For_Non-Programmers.html)
- Step-by-Step Guide to JPA-Enabled Unit Testing (Java EE) (<http://www.sizovpoint.com/2010/01/step-by-step-guide-to-jpa-enabled-unit.html>)

Unusual software bug

Unusual software bugs are a class of software bugs that are considered exceptionally difficult to understand and repair. There are several kinds, mostly named after scientists who discovered counterintuitive things.

Bohrbug

A **bohrbug** (named after the Bohr atom model) is a bug that manifests itself consistently under a well-defined (but possibly unknown) set of conditions. Thus, in contrast with heisenbugs, a bohrbug does not disappear or alter its characteristics when it is researched. These include the easiest bugs to fix (where the nature of the problem is obvious), but also bugs that are hard to find and fix and remain in the software during the operational phase. Sometimes an error might occur only when a unique data set is entered, or unique circumstances are encountered. These kinds of bugs are often present in parts of source code that are not invoked very often and thus might remain undetected for an extended period of time, and are sometimes termed a *ghost in the code*.

For example, an overflow bug in a by-the-book binary search algorithm may exhibit itself only when the data array under search is very large and the item to be searched for is located near the end of the array. Because programmers tend to test their work using small arrays of data, and only recently have there existed machines with enough memory to hold a sufficiently large array, such a bug may go undetected for many years.^[1]

Mandelbug

A **mandelbug** (named after fractal innovator Benoît Mandelbrot) is a computer bug whose causes are so complex that its behavior appears chaotic or even non-deterministic.^[2] This word also implies that the speaker thinks it is a bohrbug rather than a heisenbug.

Mandelbug is sometimes used to describe a bug whose behavior does not appear chaotic, but whose causes are so complex that there is no practical solution. An example of this is a bug caused by a flaw in the fundamental design of the entire system.

In the literature, there are inconsistent statements about the relationships between bohrbug, heisenbug, and mandelbug: According to the above definition, mandelbugs are bohrbugs. Heisenbug and bohrbug are considered antonyms. Moreover, it is claimed that all heisenbugs are mandelbugs.^[3]

In a column in IEEE Computer,^[4] *mandelbug* is considered the complementary antonym to *bohrbug*; i.e., a software bug is either a bohrbug or a mandelbug. The apparently complex behavior of a mandelbug is assumed to be caused either by long delays between fault activation and the failure occurrence, or by influences of other software system elements (hardware, operating system, other applications) on the fault's behavior. Heisenbugs (whose behavior is influenced by a debugger, or other means of investigating the fault) are mandelbugs.

Heisenbug

A **heisenbug** (named after the Heisenberg Uncertainty Principle) is a computer bug that disappears or alters its characteristics when an attempt is made to study it.

One common example is a bug that occurs in a program that was compiled with an optimizing compiler, but not in the same program when compiled without optimization (e.g., for generating a debug-mode version). Another example is a bug caused by a race condition. A heisenbug may also appear in a system that does not conform to the command-query separation design guideline, since a routine called more than once could return different values each time, generating hard-to-reproduce bugs in a race condition scenario.

The name *heisenbug* is a pun on the Heisenberg uncertainty principle, a quantum physics concept which is commonly (yet inaccurately) used to refer to the fact that in the Copenhagen Interpretation model of quantum mechanical behaviour, observers affect what they are observing, by the mere act of observing it alone (this is actually the observer effect, and is commonly confused with the Heisenberg uncertainty principle).

One common reason for heisenbug-like behaviour is that executing a program in debug mode often cleans memory before the program starts, and forces variables onto stack locations, instead of keeping them in registers. These differences in execution can alter the effect of bugs involving out-of-bounds member access, incorrect assumptions about the initial contents of memory, or floating-point comparisons (for instance, when a floating-point variable in a 32-bit stack location is compared to one in an 80-bit register). Another reason is that debuggers commonly provide watches or other user interfaces that cause additional code (such as property accessors) to be executed, which can, in turn, change the state of the program. Yet another reason is a fandango on core, the effect of a pointer running out of bounds. Many heisenbugs are caused by uninitialized values.

Time can also be a factor in heisenbugs. Executing a program under control of a debugger can change the execution timing of the program as compared to normal execution. Time-sensitive bugs such as race conditions may not reproduce when the program is slowed down by single-stepping source lines in the debugger. This is particularly true when the behavior involves interaction with an entity not under the control of a debugger, such as when debugging network packet processing between two machines and only one is under debugger control.

In an interview Bruce Lindsay tells of being there when the term was first used, and that it was created because Heisenberg said, "The more closely you look at one thing, the less closely can you see something else."^[5]

This claim of origin is almost certainly wrong, as the term has been used for over two decades. For example, the earliest Google-archived mention is from the mailing list (later Usenet news group) comp.risks, moderated by Peter G. Neumann. In RISKS Digest Volume 4 : Issue 34, dated 23 December 1986,^[6] Zhahai Stewart contributes an item titled "Another heisenbug" noting that many such contributions have appeared in recent issues of RISKS Digest. The term, and especially the distinction Heisenbug/Bohrbug, was already mentioned in 1985 by Jim Gray in a paper about software failures.^[7]

Schrödinbug

A **schrödinbug** is a bug that manifests only after someone reading source code or using the program in an unusual way notices that it never should have worked in the first place, at which point the program promptly stops working for everybody until fixed. The Jargon File^[8] adds: "Though... this sounds impossible, it happens; some programs have harbored latent schrödinbugs for years."

The name *schrödinbug* was introduced in the version 2.9.9 of the Jargon file, published in April 1992. It is derived from the Schrödinger's cat thought experiment. A well-written program executing in a reliable computing environment is expected to follow the principle of determinism, and that being so the quantum questions of observability (i.e., breaking the program by reading the source code) posited by Schrödinger (i.e., killing the cat by opening the box) affecting the operation of a program is unexpected.

Repairing an obviously defective piece of code is often more important than determining what arcane set of circumstances caused it to work at all (or appear to work) in the first place, and why it then stopped. Because of this, many of these bugs are never fully understood. When a bug of this type is examined in enough detail, it can usually be reclassified as a bohrbug, heisenbug, or mandelbug.

Phase of the Moon bug

The phase of the moon is sometimes spouted as a silly parameter on which a bug might depend, such as when exasperated after trying to isolate the true cause. The Jargon File documents two rare instances in which data processing problems were actually caused by phase-of-the-moon timing.^[9]

In general, programs that exhibit time-dependent behavior are vulnerable to time-dependent failures. These could occur during a certain part of a scheduled process, or at special times, such as on leap days or when a process crosses a daylight saving time, day, month, year, or century boundary (as with the Year 2000 bug).

Statistical bug

Statistical bugs can only be detected in aggregates and not in single runs of a section of code. These are bugs that usually affect code that is supposed to produce random or pseudo-random output. An example is code to generate points uniformly distributed on the surface of a sphere, say, and the result is that there are significantly more points in the northern hemisphere than the southern one. Tracing in detail through a single run of the point generator can completely fail to shed light on the location of such a bug because it is impossible to identify the output of any one run as wrong – after all, it's intended to be random. Only when many points are generated does the problem become apparent. Popular debugging techniques such as checking pre- and postconditions can do little to help. Similar problems can also occur in numerical algorithms in which each individual operation is accurate to within a given tolerance but where numerical errors accumulate only after a large number of runs, especially if the errors have a systematic bias. A simple example of this is the `strfry()` function in the GNU C Library.^[10]

Alpha particle bug (single event upset)

The term alpha particle bug derives from the historical phenomenon of soft errors caused by cosmic rays. These are energetic charged subatomic particles, originating from outer space. When cosmic rays collide with molecules in the atmosphere, they produce a shower of billions of high energy radioactive particles. These particles could disturb an electron in RAM, and thus change a 0 to a 1, and vice-versa. Thus the term is used to describe a class of bug where an issue was only seen once, was verifiable at the time, but source code analysis indicates that the bug should be impossible, thus the only explanation is that an alpha particle disturbed an electron. The likely cause of such bugs is build or integration errors, or some form of unusual memory corruption. This bug is often referred to by spacecraft developers as a single event upset.

According to a study done by Intel in 1990, the number of errors caused by cosmic rays increases with the altitude of the computer and drops to zero if the computer is running in a cave.^[11] Therefore, computer chips in airplanes, space craft and other sensitive systems will have error checking in ram while common desktop computers will not. In 1998, only one error per month per 256 MiB of ram was expected for a desktop computer. However, as chip density increases, Intel expects the errors caused by cosmic rays to increase and be a limiting factor in design.^[12]

References

This article was originally based on material from the Free On-line Dictionary of Computing, which is licensed under the GFDL.

- [1] Joshua Bloch, "Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken" (<http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html>). Official Google Research Blog, June 2, 2006.
- [2] The Jargon File - mandelbug (<http://catb.org/jargon/html/M/mandelbug.html>)
- [3] [The following article investigates the various definitions of bohrbug, mandelbug and heisenbug proposed in the literature, as well as the statements made about the relationships between these fault types: M. Grottko and K. S. Trivedi, Software Faults, Software Aging and Software Rejuvenation. Journal of the Reliability Engineering Association of Japan, Vol. 27, No. 7, pp. 425-438, 2005.]
- [4] IEEE Computer vol. 40, no. 2 - February 2007 (http://www.computer.org/portal/site/computer/menuitem.5d61c1d591162e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1_article&TheCat=1065&path=computer/homepage/Feb07&file=softwaretech.xml&xsl=article.xsl)
- [5] ACM Queue vol. 2, no. 8 - November 2004 (<http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=233&page=5>)
- [6] Risks-List on Google Groups RISKS DIGEST 4.34 (http://groups.google.com/group/mod.risks/browse_thread/thread/83a6ad844eda93e0/5e061e6da0c2dbfc?lnk=st&q=heisenbug&rnum=896#5e061e6da0c2dbfc)
- [7] "Why Do Computers Stop And What Can Be Done About It?" (<http://citeseer.ist.psu.edu/gray85why.html>). 1985. .
- [8] <http://www.catb.org/jargon/html/S/schroedinbug.html>
- [9] CATB.org, "phase of the moon" (<http://www.catb.org/jargon/html/P/phase-of-the-moon.html>)
- [10] strfry() gives skewed distributions (http://sourceware.org/bugzilla/show_bug.cgi?id=4403)
- [11] http://findarticles.com/p/articles/mi_qn4158/is_19980728/ai_n14164062
- [12] <http://www.newscientist.com/blog/technology/2008/03/do-we-need-cosmic-ray-alerts-for.html>

External links

- The Jargon File has entries on heisenbug (<http://www.catb.org/jargon/html/H/heisenbug.html>), Bohr bug (<http://www.catb.org/jargon/html/B/Bohr-bug.html>), mandelbug (<http://www.catb.org/jargon/html/M/mandelbug.html>) and schroedinbug (<http://www.catb.org/jargon/html/S/schroedinbug.html>), and phase of the moon (<http://www.catb.org/jargon/html/P/phase-of-the-moon.html>)
- The Heisenberg Debugging Technology (<http://sourceware.org/gdb/talks/esc-west-1999/>)
- Testing parallel programs (<http://multicore.ning.com/profiles/blogs/testing-parallel-programs>)
- A Story About Magic (<http://ftp.sunet.se/jargon/html/magic-story.html>)

Usability testing

Usability testing is a technique used to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system.^[1] This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users.

Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are foods, consumer products, web sites or web applications, computer interfaces, documents, and devices. Usability testing measures the usability, or ease of use, of a specific object or set of objects, whereas general human-computer interaction studies attempt to formulate universal principles.

History of usability testing

Henry Dreyfuss in the late 1940s contracted to design the state rooms for the twin ocean liners "Independence" and "Constitution." He built eight prototype staterooms and installed them in a warehouse. He then brought in a series of travelers to "live" in the rooms for a short time, bringing with them all items they would normally take when cruising. His people were able to discover over time, for example, if there was space for large steamer trunks, if light switches needed to be added beside the beds to prevent injury, etc., before hundreds of state rooms had been built into the ship.^[2]

A Xerox Palo Alto Research Center (PARC) employee wrote that PARC used extensive usability testing in creating the Xerox Star, introduced in 1981.^[3] Only about 25,000 were sold, leading many to consider the Xerox Star a commercial failure.

The Inside Intuit book, says (page 22, 1984), "... in the first instance of the Usability Testing that later became standard industry practice, LeFevre recruited people off the streets... and timed their Kwik-Chek (Quicken) usage with a stopwatch. After every test... programmers worked to improve the program."^[4] Scott Cook, Intuit co-founder, said, "... we did usability testing in 1984, five years before anyone else... there's a very big difference between doing it and having marketing people doing it as part of their... design... a very big difference between doing it and having it be the core of what engineers focus on."^[5]

Goals of usability testing

Usability testing is a black-box testing technique. The aim is to observe people using the product to discover errors and areas of improvement. Usability testing generally involves measuring how well test subjects respond in four areas: efficiency, accuracy, recall, and emotional response. The results of the first test can be treated as a baseline or control measurement; all subsequent tests can then be compared to the baseline to indicate improvement.

- *Performance* -- How much time, and how many steps, are required for people to complete basic tasks? (For example, find something to buy, create a new account, and order the item.)
 - *Accuracy* -- How many mistakes did people make? (And were they fatal or recoverable with the right information?)
 - *Recall* -- How much does the person remember afterwards or after periods of non-use?
 - *Emotional response* -- How does the person feel about the tasks completed? Is the person confident, stressed? Would the user recommend this system to a friend?
-

What usability testing is not

Simply gathering opinions on an object or document is market research or qualitative research rather than usability testing. Usability testing usually involves systematic observation under controlled conditions to determine how well people can use the product.^[6] However, often both qualitative and usability testing are used in combination, to better understand users' motivations/perceptions, in addition to their actions.

Rather than showing users a rough draft and asking, "Do you understand this?", usability testing involves watching people trying to *use* something for its intended purpose. For example, when testing instructions for assembling a toy, the test subjects should be given the instructions and a box of parts and, rather than being asked to comment on the parts and materials, they are asked to put the toy together. Instruction phrasing, illustration quality, and the toy's design all affect the assembly process.

Methods

Setting up a usability test involves carefully creating a scenario, or realistic situation, wherein the person performs a list of tasks using the product being tested while observers watch and take notes. Several other test instruments such as scripted instructions, paper prototypes, and pre- and post-test questionnaires are also used to gather feedback on the product being tested. For example, to test the attachment function of an e-mail program, a scenario would describe a situation where a person needs to send an e-mail attachment, and ask him or her to undertake this task. The aim is to observe how people function in a realistic manner, so that developers can see problem areas, and what people like. Techniques popularly used to gather data during a usability test include think aloud protocol, Co-discovery Learning and eye tracking.

Hallway testing

Hallway testing (or **Hall Intercept Testing**) is a general methodology of usability testing. Rather than using an in-house, trained group of testers, just five to six random people, indicative of a cross-section of end users, are brought in to test the product, or service. The name of the technique refers to the fact that the testers should be random people who pass by in the hallway.^[7]

Hallway testing is particularly effective in the early stages of a new design when the designers are looking for "brick walls," problems so serious that users simply cannot advance. Anyone of normal intelligence other than designers and engineers can be used at this point. (Both designers and engineers immediately turn from being test subjects into being "expert reviewers." They are often too close to the project, so they already know how to accomplish the task, thereby missing ambiguities and false paths.)

Remote Usability Testing

In a scenario where usability evaluators, developers and prospective users are located in different countries and time zones, conducting a traditional lab usability evaluation creates challenges both from the cost and logistical perspectives. These concerns led to research on remote usability evaluation, with the user and the evaluators separated over space and time. Remote testing, which facilitates evaluations being done in the context of the user's other tasks and technology can be either synchronous or asynchronous. Synchronous usability testing methodologies involve video conferencing or employ remote application sharing tools such as WebEx. The former involves real time one-on-one communication between the evaluator and the user, while the latter involves the evaluator and user working separately.^[8]

Asynchronous methodologies include automatic collection of user's click streams, user logs of critical incidents that occur while interacting with the application and subjective feedback on the interface by users.^[9] Similar to an in-lab study, an asynchronous remote usability test is task-based and the platforms allow you to capture clicks and task times. Hence, for many large companies this allows you to understand the WHY behind the visitors' intents when

visiting a website or mobile site. Additionally, this style of user testing also provides an opportunity to segment feedback by demographic, attitudinal and behavioural type. The tests are carried out in the user's own environment (rather than labs) helping further simulate real-life scenario testing. This approach also provides a vehicle to easily solicit feedback from users in remote areas.

Numerous tools are available to address the needs of both these approaches. WebEx and Go-to-meeting are the most commonly used technologies to conduct a synchronous remote usability test.^[10] However, synchronous remote testing may lack the immediacy and sense of "presence" desired to support a collaborative testing process. Moreover, managing inter-personal dynamics across cultural and linguistic barriers may require approaches sensitive to the cultures involved. Other disadvantages include having reduced control over the testing environment and the distractions and interruptions experienced by the participants' in their native environment.^[11] One of the newer methods developed for conducting a synchronous remote usability test is by using virtual worlds.^[12]

Expert review

Expert review is another general method of usability testing. As the name suggests, this method relies on bringing in experts with experience in the field (possibly from companies that specialize in usability testing) to evaluate the usability of a product.

Automated expert review

Similar to expert reviews, **automated expert reviews** provide usability testing but through the use of programs given rules for good design and heuristics. Though an automated review might not provide as much detail and insight as reviews from people, they can be finished more quickly and consistently. The idea of creating surrogate users for usability testing is an ambitious direction for the Artificial Intelligence community.

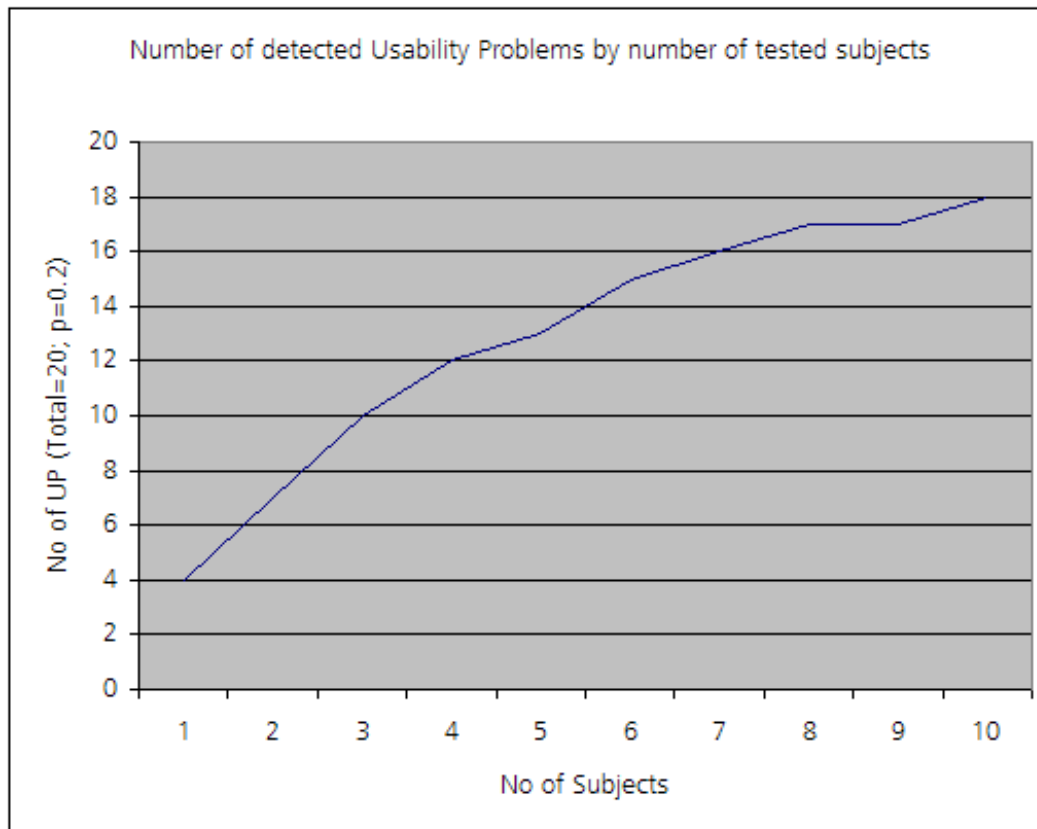
How many users to test?

In the early 1990s, Jakob Nielsen, at that time a researcher at Sun Microsystems, popularized the concept of using numerous small usability tests—typically with only five test subjects each—at various stages of the development process. His argument is that, once it is found that two or three people are totally confused by the home page, little is gained by watching more people suffer through the same flawed design. "Elaborate usability tests are a waste of resources. The best results come from testing no more than five users and running as many small tests as you can afford."^[7] Nielsen subsequently published his research and coined the term heuristic evaluation.

The claim of "Five users is enough" was later described by a mathematical model^[13] which states for the proportion of uncovered problems U

$$U = 1 - (1 - p)^n$$

where p is the probability of one subject identifying a specific problem and n the number of subjects (or test sessions). This model shows up as an asymptotic graph towards the number of real existing problems (see figure below).



In later research Nielsen's claim has eagerly been questioned with both empirical evidence^[14] and more advanced mathematical models.^[15] Two key challenges to this assertion are:

1. since usability is related to the specific set of users, such a small sample size is unlikely to be representative of the total population so the data from such a small sample is more likely to reflect the sample group than the population they may represent
2. Not every usability problem is equally easy-to-detect. Intractable problems happen to decelerate the overall process. Under these circumstances the progress of the process is much shallower than predicted by the Nielsen/Landauer formula.^[16]

It is worth noting that Nielsen does not advocate stopping after a single test with five users; his point is that testing with five users, fixing the problems they uncover, and then testing the revised site with five different users is a better use of limited resources than running a single usability test with 10 users. In practice, the tests are run once or twice per week during the entire development cycle, using three to five test subjects per round, and with the results delivered within 24 hours to the designers. The number of users actually tested over the course of the project can thus easily reach 50 to 100 people.

In the early stage, when users are most likely to immediately encounter problems that stop them in their tracks, almost anyone of normal intelligence can be used as a test subject. In stage two, testers will recruit test subjects across a broad spectrum of abilities. For example, in one study, experienced users showed no problem using any design, from the first to the last, while naive user and self-identified power users both failed repeatedly.^[17] Later on, as the design smooths out, users should be recruited from the target population.

When the method is applied to a sufficient number of people over the course of a project, the objections raised above become addressed: The sample size ceases to be small and usability problems that arise with only occasional users are found. The value of the method lies in the fact that specific design problems, once encountered, are never seen again because they are immediately eliminated, while the parts that appear successful are tested over and over. While it's true that the initial problems in the design may be tested by only five users, when the method is properly applied, the parts of the design that worked in that initial test will go on to be tested by 50 to 100 people.

References


- [1] Nielsen, J. (1994). *Usability Engineering*, Academic Press Inc, p 165
- [2] NN/G Usability Week 2011 Conference "Interaction Design" Manual, Bruce Tognazzini, Nielsen Norman Group, 2011
- [3] <http://interactions.acm.org/content/XV/baecker.pdf>
- [4] http://books.google.com/books?id=IRs_4U43UcEC&printsec=frontcover&sig=ACfU3U1xvA7-f80TP9Zqt9wkB9adVAqZ4g#PPA22,M1
- [5] <http://news.zdnet.co.uk/itmanagement/0,1000000308,2065537,00.htm>
- [6] <http://jerz.setonhill.edu/design/usability/intro.htm>
- [7] <http://www.useit.com/alertbox/20000319.html>
- [8] <http://portal.acm.org/citation.cfm?id=1240838&dl=>
- [9] <http://portal.acm.org/citation.cfm?id=971264>
- [10] http://www.boxesandarrows.com/view/remote_online_usability_testing_why_how_and_when_to_use_it
- [11] Dray, Susan; Siegel, David (March 2004). "Remote possibilities?: international usability testing at a distance". *Interactions* **11** (2): 10–17. doi:10.1145/971258.971264.
- [12] Chalil Madathil, Kapil; Joel S. Greenstein (May 2011). "Synchronous remote usability testing: a new approach facilitated by virtual worlds". *Proceedings of the 2011 annual conference on Human factors in computing systems*. CHI '11: 2225–2234. doi:<http://portal.acm.org/citation.cfm?doid=1978942.1979267>.
- [13] Virzi, R.A., Refining the Test Phase of Usability Evaluation: How Many Subjects is Enough? *Human Factors*, 1992. 34(4): p. 457–468.
- [14] <http://citeseer.ist.psu.edu/spool01testing.html>
- [15] Caulton, D.A., Relaxing the homogeneity assumption in usability testing. *Behaviour & Information Technology*, 2001. 20(1): p. 1-7
- [16] Schmettow, Heterogeneity in the Usability Evaluation Process. In: M. England, D. & Beale, R. (ed.), *Proceedings of the HCI 2008*, British Computing Society, 2008, 1, 89-98
- [17] Bruce Tognazzini. "Maximizing Windows" (<http://www.asktog.com/columns/000maxsrns.html>). .

External links

- Usability.gov (<http://www.usability.gov/>)
- A Brief History of the Magic Number 5 in Usability Testing (<http://www.measuringusability.com/blog/five-history.php>)

Utest

uTest, Inc.

	
Type	Private
Industry	Software Testing Crowdsourcing Software Quality Assurance
Founded	August, 2007
Headquarters	Southborough, Massachusetts
Key people	Doron Reuveni, CEO Roy Soloman, VP of Product Marc Weinstein, VP of Sales Matt Johnston, VP of Marketing John Montgomery, VP of Project Delivery Reuven Fein-Barsegian, VP of Engineering ^[1]
Products	uTest Platform
Employees	20+
Website	http://www.utest.com/

uTest, Inc. is a venture-funded^[2] software testing marketplace based in Southborough, Massachusetts.

History

The company was incorporated in August 2007^[3] by founders Doron Reuveni^[4] and Roy Solomon. It was backed by venture capital firms Longworth Partners^[5] and Egan Managed Capital^{[6] [7]}.

Products and services

uTest offers functional, usability and load & performance testing^[8] for web, mobile and desktop software applications^{[9] [10]}.

Strategy

uTest's business model is based on the idea that crowdsourcing is better suited to web and mobile app testing than other outsourcing models^[11]. With crowdsourced testing^[12], the crowd reflects the diversity (e.g. multiple geographic locations, languages spoken) of the apps and users themselves^[13]. The uTest community is made up of independent software testers who test applications across:

- Locations
- Languages
- Operating system
- Internet browsers
- Plug-ins and anti-virus programs
- Handset makers, models and wireless carriers (for mobile applications)

How it works

1. Specify testing requirements (by location, OS, browser, app type, etc.)^[14]
2. Upload testing scripts through a secure platform
3. View bugs reported by the community
4. Approve, reject or request more information from testers regarding bugs
5. Pay for each completed test cycle^[15]

uTest's platform can be integrated with bug-tracking systems such as Jira, Rally and Bugzilla.

Bug Battle

uTest holds software testing competitions each quarter, called "Bug Battles"^[16], where testers from around the world compete for cash prizes by reporting bugs in popular software applications. The company's first Bug Battle occurred in November 2008; the 1,331 software testers who participated reported more than 700 bugs in Google Chrome, Internet Explorer and Mozilla Firefox^[17]. The second Bug Battle took place in March 2009; the 1,119 software testers who participated reported bugs in Facebook, MySpace and LinkedIn^[18]. Twitter applications were the subject of the third Bug Battle in June 2009^[19]. Search engines, including Google, Google Caffeine, Bing and Yahoo, were the subjects of the fourth Bug Battle in August 2009^[20]. Most recently, the fifth Bug Battle in December 2009 - Battle of the E-Tailers^[21] - compared Amazon.com, Wal-Mart.com and Target.com. The study was featured in USA Today^[22] and Fast Company^[23]. Most recently, the Bug Battle in June 2010 - The Check-In Challenge^[24] - compared Foursquare, Gowalla and Brightkite. The report was featured in Mashable^[25] and ZDNet^[26].

External links

- uTest^[27]
- Software Testing Blog^[28]
- Lessons from 10 Recession Startups^[29], by Melanie Lindner, "Forbes", April 4, 2009.
- Crowdsourcing: What It Means for Innovation^[30], by John Winsor, "Business Week", June 15, 2009.
- Crowdsourcing: Now With a Real Business Model!^[31], by Jeff Howe, "Wired Magazine", December 2, 2008.
- The Crowd Is Wise (When It's Focused)^[32], by Steve Lohr, "The New York Times", July 18, 2009.
- Crowdsourcing Pioneer: Doron Reuveni^[33], by Andrew Muns, "Software Testing & Performance Magazine", September 1, 2009.
- Crowdsourcing: 5 Reasons It's Not Just For Startups Any More^[34], by Dion Hinchcliffe, "www.ebizQ.com", September 25, 2009.
- Study: Amazon Most User Friendly^[35], by Byron Acohido, "USA Today", December 6, 2009.
- Bug Testers: Google is Clean, Bing is Buggy^[36], by Tom Krazit, "CNET", September 15, 2009.
- Amazon Trumps Target, Wal-Mart in Bug Battle^[37], by Kate Rockwood, "Fast Company", December 7, 2009.
- Crowdsourcing: How Companies Can Launch Higher-Quality Web and Mobile Applications^[38], by Doron Reuveni, CEO of uTest, "ebizQ.com", January 4, 2010.
- To Crowdsourcing Friends, Foes & Fanatics: Just How Loyal Is Your Community?^[39], by Matt Johnston of uTest, "VentureFizz", May 25, 2010.

References

- [1] "uTest Management" (<http://www.utest.com/about-us/utest-management>). uTest, Inc.. Retrieved 2009-05-13.
- [2] uTest Raises \$5 Million More For Crowdsourced Bug Testing; TechCrunch; December 1, 2008: <http://www.techcrunch.com/2008/12/01/utest-raises-5-million-more-for-crowdsourced-bug-testing/>
- [3] About uTest: <http://www.utest.com/company>
- [4] A Crowdsourcing Pioneer; Software Testing & Performance Magazine; September 1, 2009: <http://stpcollaborative.com/knowledge/379-a-crowdsourcing-pioneer>
- [5] Longworth Venture Partners; Longworth Portfolio Companies: <http://www.longworth.com/portfolio/portfolio.html>
- [6] Egan Managed Capital; Egan Portfolio Companies: <http://www.egancapital.com/portfolio/current.php>
- [7] uTest Secures 5 million in Venture Capital Funding; Boston Globe; December 2, 2008: http://www.boston.com/business/ticker/2008/12/utest_secures_5.html
- [8] Types of Testing: <http://www.utest.com/types-testing>
- [9] What We Test: <http://www.utest.com/what-we-test>
- [10] uTest Named 2009 "Cool Vendor" by Leading Business Analyst Firm, Gartner; May 31, 2009: http://software.dbusinessnews.com/shownews.php?newsid=184124&type_news=latest
- [11] Crowdsourcing: Now with a Real Business Model; Wired Magazine; December 2, 2008: <http://www.wired.com/epicenter/2008/12/crowdsourcing-n/>
- [12] Crowdsourcing: What it Means for Innovation; BusinessWeek Magazine; June 15, 2009: http://www.businessweek.com/innovate/content/jun2009/id20090615_946326.htm
- [13] Crowdsourcing: 5 Reasons It's Not Just For Startups Any More; www.ebizQ.com; September 25, 2009: http://www.ebizq.net/blogs/enterprise/2009/09/crowdsourcing_5_reasons_its_no.php
- [14] How It Works: <http://www.utest.com/how-it-works/agile-testing>
- [15] On-Demand Pricing Model: <http://www.utest.com/pricing>
- [16] Bug Battle: <http://www.utest.com/bugbattle>
- [17] Battle of Browser Bugs; About.com; November 4, 2008: <http://browsers.about.com/b/2008/11/04/battle-of-the-browser-bugs.htm>
- [18] uTest Bug Battle: Which Social Network Is The Buggiest?; TechCrunch; March 16, 2009: <http://www.techcrunch.com/2009/03/16/utest-bug-battle-which-social-network-is-the-buggiest/>
- [19] Study: Which Twitter Desktop App is Most Usable?; Mashable; June 23, 2009: <http://mashable.com/2009/06/23/twitter-app-usability/>
- [20] Bug testers: Google is clean, Bing is buggy; CNET; September 15, 2009: http://news.cnet.com/8301-30684_3-10353495-265.html
- [21] Battle of the E-Tailers: <http://www.utest.com/bugbattle/q409>
- [22] Study: Amazon most user friendly; USA Today; December 6, 2009: http://www.usatoday.com/MONEY/usaedition/2009-12-07-techlive07_ST_U.htm?csp=34
- [23] Amazon Trumps Target, Wal-Mart in Bug Battle; Fast Company; December 7, 2009: <http://www.fastcompany.com/blog/kate-rockwood/bizzy-body/amazon-trumps-target-walmart-bug-battle?partner=rss>
- [24] Battle of the Check-In Services: <http://www.utest.com/bugbattle/q210>
- [25] Which App Does Checkins Best?; Mashable; June 10, 2010; <http://mashable.com/2010/06/10/utest-check-in-challenge/>
- [26] Study reveals bug factor in Foursquare, Gowalla and Brightkite; ZDNet; June 16, 2010; http://www.zdnet.com/blog/feeds/study-reveals-bug-factor-in-foursquare-gowalla-and-brightkite/2859?utm_source=twitterfeed&utm_medium=twitter&utm_campaign=Feed%3A+zdnet%2Ffeeds+%28ZDNet+Feeds%29
- [27] <http://www.utest.com/>
- [28] <http://blog.utest.com/>
- [29] http://www.forbes.com/2009/04/07/recession-startup-obstacles-entrepreneurs-management-startup_slide_6.html?thisSpeed=40000/
- [30] http://www.businessweek.com/innovate/content/jun2009/id20090615_946326.htm/
- [31] <http://www.wired.com/epicenter/2008/12/crowdsourcing-n/>
- [32] <http://www.nytimes.com/2009/07/19/technology/internet/19unboxed.html/>
- [33] <http://stpcollaborative.com/knowledge/379-a-crowdsourcing-pioneer/A>
- [34] http://www.ebizq.net/blogs/enterprise/2009/09/crowdsourcing_5_reasons_its_no.php/
- [35] http://www.usatoday.com/MONEY/usaedition/2009-12-07-techlive07_ST_U.htm?csp=34/
- [36] http://news.cnet.com/8301-30684_3-10353495-265.html/
- [37] <http://www.fastcompany.com/blog/kate-rockwood/bizzy-body/amazon-trumps-target-walmart-bug-battle?partner=rss/>
- [38] <http://www.ebizq.net/topics/collaboration/features/12078.html/>
- [39] <http://venturefizz.com/blog/crowdsourcing-friends-foes-fanatics-just-how-loyal-your-community/>

Verification and Validation (software)

In software project management, software testing, and software engineering, **Verification and Validation (V&V)** is the process of checking that a software system meets specifications and that it fulfills its intended purpose. It is normally part of the software testing process of a project.

Definitions

Also known as software quality control.

Validation checks that the product design satisfies or fits the intended usage (high-level checking) — i.e., you built the right product. This is done through dynamic testing and other forms of review.

According to the Capability Maturity Model (CMMI-SW v1.1),

- **Verification:** The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [IEEE-STD-610].
- **Validation:** The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. [IEEE-STD-610]

In other words, validation ensures that the product actually meets the user's needs, and that the specifications were correct in the first place, while verification is ensuring that the product has been built according to the requirements and design specifications. Validation ensures that 'you built the right thing'. Verification ensures that 'you built it right'. Validation confirms that the product, as provided, will fulfill its intended use.

From testing perspective:

- **Fault** - wrong or missing function in the code.
- **Failure** - the manifestation of a fault during execution.
- **Malfunction** - according to its specification the system does not meet its specified functionality.

Within the modeling and simulation community, the definitions of validation, verification and accreditation are similar:

- Validation is the process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated data are accurate representations of the real world from the perspective of the intended use(s).^[1]
- Accreditation is the formal certification that a model or simulation is acceptable to be used for a specific purpose.^[1]
- Verification is the process of determining that a computer model, simulation, or federation of models and simulations implementations and their associated data accurately represents the developer's conceptual description and specifications.^[1]

Related concepts

Both verification and validation are related to the concepts of quality and of software quality assurance. By themselves, verification and validation do not guarantee software quality; planning, traceability, configuration management and other aspects of software engineering are required.

Classification of methods

In mission-critical systems where flawless performance is absolutely necessary, formal methods can be used to ensure the correct operation of a system. However, often for non-mission-critical systems, formal methods prove to be very costly and an alternative method of V&V must be sought out. In this case, syntactic methods are often used.

Test cases

A test case is a tool used in the process.

Test cases are prepared for verification: to determine if the process that was followed to develop the final product is right.

Test case are executed for validation: if the product is built according to the requirements of the user. Other methods, such as reviews, are used when used early in the Software Development Life Cycle provide for validation.

Independent Verification and Validation

Verification and validation often is carried out by a separate group from the development team; in this case, the process is called "**Independent Verification and Validation**", or **IV&V**.

Regulatory environment

Verification and validation must meet the compliance requirements of law regulated industries, which is often guided by government agencies^[2] ^[3] or industrial administrative authorities. e.g. The FDA requires software versions and patches to be validated.^[4]

Notes and references

- [1] *Department of Defense Documentation of Verification, Validation & Accreditation (VV&A) for Models and Simulations*. Missile Defense Agency. 2008
 - [2] "General Principles of Software validation; Final Guidance for Industry and FDA Staff" (<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085371.pdf>) (PDF). Food and Drug Administration. 11 January 2002. . Retrieved 12 July 2009.
 - [3] "Guidance for Industry: Part 11, Electronic Records; Electronic Signatures — Scope and Application" (<http://www.fda.gov/downloads/Drugs/GuidanceComplianceRegulatoryInformation/Guidances/UCM072322.pdf>) (PDF). Food and Drug Administration. August 2003. . Retrieved 12 July 2009.
 - [4] "Guidance for Industry: Cybersecurity for Networked Medical Devices Containing Off-the Shelf (OTS) Software" (<http://www.fda.gov/downloads/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm077823.pdf>). Food and Drug Administration. 14 January 2005. . Retrieved 12 July 2009.
- Tran, E. (1999). "Verification/Validation/Certification" (http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html). In Koopman, P.. *Topics in Dependable Embedded Systems*. Carnegie Mellon University. Retrieved 2007-05-18.
 - Menzies, T.; Y. Hu (2003). "Data mining for very busy people". *IEEE Computer* **36** (1): 22–29. doi:10.1109/MC.2003.1244531.

External links

- Chapter on Software quality (including VnV) (<http://www.computer.org/portal/web/swebok/html/ch11>) in SWEBOK

Volume testing

Volume Testing belongs to the group of non-functional tests, which are often misunderstood and/or used interchangeably. Volume testing refers to testing a software application with a certain amount of data. This amount can, in generic terms, be the database size or it could also be the size of an interface file that is the subject of volume testing. For example, if you want to volume test your application with a specific database size, you will expand your database to that size and then test the application's performance on it. Another example could be when there is a requirement for your application to interact with an interface file (could be any file such as .dat, .xml); this interaction could be reading and/or writing on to/from the file. You will create a sample file of the size you want and then test the application's functionality with that file in order to test the performance.

Vulnerability (computing)

In computer security, a **vulnerability** is a weakness which allows an attacker to reduce a system's information assurance.

Vulnerability is the intersection of three elements: a system susceptibility or flaw, attacker access to the flaw, and attacker capability to exploit the flaw.^[1] To be vulnerable, an attacker must have at least one applicable tool or technique that can connect to a system weakness. In this frame, vulnerability is also known as the attack surface.

Vulnerability management is the cyclical practice of identifying, classifying, remediating, and mitigating vulnerabilities^[2] This practice generally refers to software vulnerabilities in computing systems.

A security risk may be classified as a vulnerability. The usage of vulnerability with the same meaning of risk can lead to confusion. The risk is tied to the potential of a significant loss. Then there are vulnerabilities without risk: for example when the affected asset has no value. A vulnerability with one or more known instances of working and fully implemented attacks is classified as an exploitable vulnerability - a vulnerability for which an exploit exists. The **window of vulnerability** is the time from when the security hole was introduced or manifested in deployed software, to when access was removed, a security fix was available/deployed, or the attacker was disabled.

Security bug is a narrower concept: there are vulnerabilities that are not related to software: hardware, site, personnel vulnerabilities are examples of vulnerabilities that are not security software bugs.

Constructs in programming languages that are difficult to use properly can be a large source of vulnerabilities.

Definitions

ISO 27005 defines **vulnerability** as:^[3]

A weakness of an asset or group of assets that can be exploited by one or more threats

where an *asset* is anything that can has value to the organization, its business operations and their continuity, including information resources that support the organization's mission^[4]

IETF RFC 2828 define **vulnerability** as:^[5]

A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy

The Committee on National Security Systems of United States of America defined **vulnerability** in CNSS Instruction No. 4009 dated 26 April 2010 National Information Assurance Glossary^[6] :

Vulnerability - Weakness in an IS, system security procedures, internal controls, or implementation that could be exploited

Many NIST publications define **vulnerability** in IT contest in different publications: FISMApedia ^[7] term ^[8] provide a list. Between them SP 800-30,^[9] give a broader one:

A flaw or weakness in system security procedures, design, implementation, or internal controls that could be exercised (accidentally triggered or intentionally exploited) and result in a security breach or a violation of the system's security policy.

ENISA defines **vulnerability** in ^[10] as:

The existence of a weakness, design, or implementation error that can lead to an unexpected, undesirable event [G.11] compromising the security of the computer system, network, application, or protocol involved.(ITSEC)

The Open Group defines **vulnerability** in ^[11] as:

The probability that threat capability exceeds the ability to resist the threat.

Factor Analysis of Information Risk (FAIR) defines **vulnerability** as ^[12]:

The probability that an asset will be unable to resist the actions of a threat agent

According FAIR vulnerability is related to Control Strength, i.e. the strength of a control as compared to a standard measure of force and the threat Capabilities, i.e. the probable level of force that a threat agent is capable of applying against an asset.

ISACA defines **vulnerability** in Risk It framework as:

A weakness in design, implementation, operation or internal control

Data and Computer Security: Dictionary of standards concepts and terms, authors Dennis Longley and Michael Shain, Stockton Press, ISBN 0-935859-17-9, defines **vulnerability** as:

1) In computer security, a weakness in automated systems security procedures, administrative controls, Internet controls, etc., that could be exploited by a threat to gain unauthorized access to information of to disrupt critical processing. 2) In computer security, a weakness in the physical layout, organization, procedures, personnel, management, administration, hardware or software that may be exploited to cause harm to the ADP system or activity. 3) In computer security, any weakness or flaw existing in a system. The attack or harmful event, or the opportunity available to a threat agent to mount that attack.

Matt Bishop and Dave Bailey ^[13] give the following definition of computer **vulnerability**:

A computer system is composed of states describing the current configuration of the entities that make up the computer system. The system computes through the application of state transitions that change the state of the system. All states reachable from a given initial state using a set of state transitions fall into the class of authorized or unauthorized, as defined by a security policy. In this paper, the definitions of these classes and transitions is considered axiomatic. A vulnerable state is an authorized state from which an unauthorized state can be reached using authorized state transitions. A compromised state is the state so reached. An attack is a sequence of authorized state transitions which end in a compromised state. By definition, an attack begins in a vulnerable state. A vulnerability is a characterization of a vulnerable state which distinguishes it from all non-vulnerable states. If generic, the vulnerability may characterize many vulnerable states; if specific, it may characterize only one...

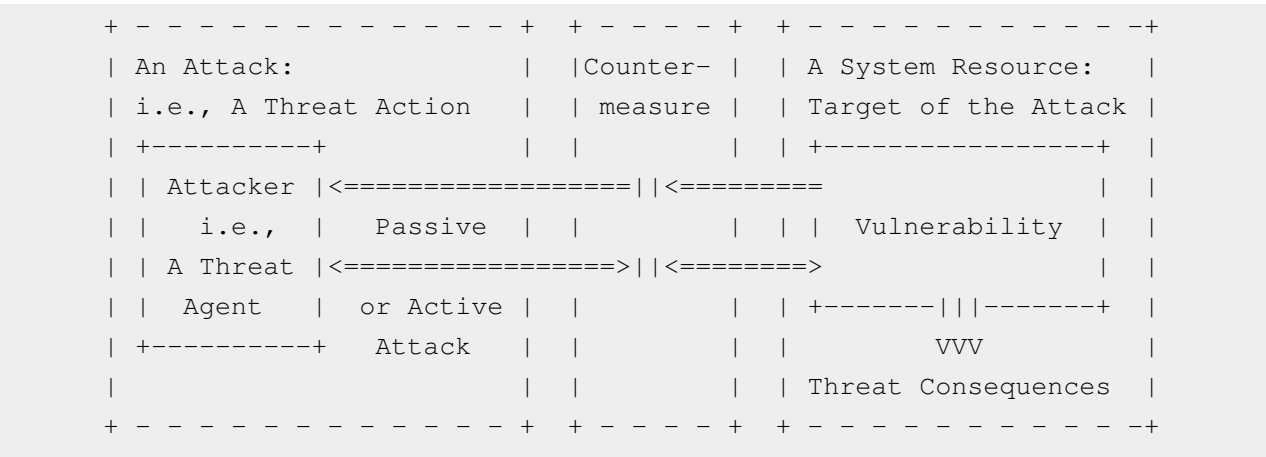
National Information Assurance Training and Education Center defines **vulnerability**: ^[14] ^[15]

A weakness in automated system security procedures, administrative controls, internal controls, and so forth, that could be exploited by a threat to gain unauthorized access to information or disrupt critical processing. 2. A weakness in system security procedures, hardware design, internal controls, etc. , which could be exploited to gain unauthorized access to classified or sensitive information. 3. A weakness in the physical layout, organization, procedures, personnel, management, administration, hardware, or software that may be exploited to cause harm to the ADP system or activity. The presence of a vulnerability does not in itself cause

harm; a vulnerability is merely a condition or set of conditions that may allow the ADP system or activity to be harmed by an attack. 4. An assertion primarily concerning entities of the internal environment (assets); we say that an asset (or class of assets) is vulnerable (in some way, possibly involving an agent or collection of agents); we write: $V(i,e)$ where: e may be an empty set. 5. Susceptibility to various threats. 6. A set of properties of a specific internal entity that, in union with a set of properties of a specific external entity, implies a risk. 7. The characteristics of a system which cause it to suffer a definite degradation (incapability to perform the designated mission) as a result of having been subjected to a certain level of effects in an unnatural (manmade) hostile environment.

Phenomenology

The term "vulnerability" relates to some other basic security terms as shown in the following diagram.^[5]

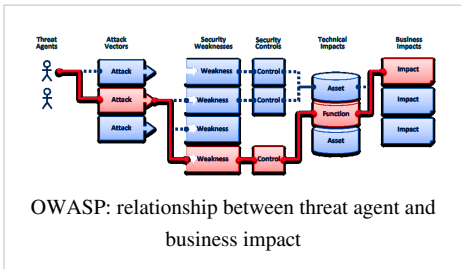


A resource (either physical or logical) may have one or more vulnerabilities that can be exploited by a threat agent in a threat action. The result can potentially compromise the confidentiality, integrity or availability of resources (not necessarily the vulnerable one) belonging to an organization and/or others parties involved(customers, suppliers).

The so called CIA triad is the basis of Information Security.

The attack can be *active* when it attempts to alter system resources or affect their operation: so it compromises integrity or availability. A *passive attack* attempts to learn or make use of information from the system but does not affect system resources: so it compromises Confidentiality.^[5]

OWASP (see figure) depicts the same phenomenon in slightly different terms: a threat agent through an attack vector exploits a weakness (vulnerability) of the system and the related security controls causing an technical impact on an IT resource (asset) connected to a business impact.



A set of policies concerned with information security management, the Information Security Management Systems (ISMS), has been developed to manage, according to Risk management principles, the countermeasures in order to accomplish to a security strategy set up following rules and regulations applicable in a country. Countermeasures are also called Security controls; when applied to the transmission of information are named security services.^[16]

The overall picture represents the risk factors of the risk scenario.^[17]

Classification

Vulnerabilities are classified according to the asset class they related to:^[3]

- hardware
 - susceptibility to humidity
 - susceptibility to dust
 - susceptibility to soiling
 - susceptibility to unprotected storage
- software
 - insufficient testing
 - lack of audit trail
- network
 - unprotected communication lines
 - insecure network architecture
- personnel
 - inadequate recruiting process
 - inadequate security awareness
- site
 - area subject to flood
 - unreliable power source
- organizational
 - lack of regular audits
 - lack of continuity plans

Causes

- Complexity: Large, complex systems increase the probability of flaws and unintended access points ^[18]
 - Familiarity: Using common, well-known code, software, operating systems, and/or hardware increases the probability an attacker has or can find the knowledge and tools to exploit the flaw ^[19]
 - Connectivity: More physical connections, privileges, ports, protocols, and services and time each of those are accessible increase vulnerability ^[12]
 - Password management flaws: The computer user uses weak passwords that could be discovered by brute force. The computer user stores the password on the computer where a program can access it. Users re-use passwords between many programs and websites. ^[18]
 - Fundamental operating system design flaws: The operating system designer chooses to enforce suboptimal policies on user/program management. For example operating systems with policies such as default permit grant every program and every user full access to the entire computer. ^[18] This operating system flaw allows viruses and malware to execute commands on behalf of the administrator. [20]
 - Internet Website Browsing: Some internet websites may contain harmful Spyware or Adware that can be installed automatically on the computer systems. After visiting those websites, the computer systems become infected and personal information will be collected and passed on to third party individuals. ^[21]
 - Software bugs: The programmer leaves an exploitable bug in a software program. The software bug may allow an attacker to misuse an application. ^[18]
 - Unchecked user input: The program assumes that all user input is safe. Programs that do not check user input can allow unintended direct execution of commands or SQL statements (known as Buffer overflows, SQL injection or other non-validated inputs). ^[18]
-

- Not learning from past mistakes: ^{[22] [23]} for example most vulnerabilities discovered in IPv4 protocol software were discovered in the new IPv6 implementations ^[24]

The research has shown that the most vulnerable point in most information systems is the human user, operator, designer, or other human: ^[25] so humans should be considered in their different roles as asset, threat, information resources. Social engineering is an increasing security concern.

Vulnerability consequences

The impact of a security breach can be very high. The fact that IT managers, or upper management, can (easily) know that IT systems and applications have vulnerabilities and do not perform any action to manage the IT risk is seen as a misconduct in most legislations. Privacy law forces managers to act to reduce the impact or likelihood that security risk. Information technology security audit is a way to let other independent people certify that the IT environment is managed properly and lessen the responsibilities, at least having demonstrated the good faith. Penetration test is a form of verification of the weakness and countermeasures adopted by an organization: a White hat hacker tries to attack an organization information technology assets, to find out how is easy or difficult to compromise the IT security. ^[26] The proper way to professionally manage the IT risk is to adopt an Information Security Management System, such as ISO/IEC 27002 or Risk IT and follow them, according to the security strategy set forth by the upper management. ^[16]

One of the key concept of information security is the principle of defence in depth: i.e. to set up a multilayer defence system that can:

- prevent the exploit
- detect and intercept the attack
- find out the threat agents and persecute them

Intrusion detection system is an example of a class of systems used to detect attacks.

Physical security is a set of measures to protect physically the information asset: if somebody can get physical access to the information asset is quite easy to made resources unavailable to its legitimate users.

Some set of criteria to be satisfied by a computer, its operating system and applications in order to meet a good security level have been developed: ITSEC and Common criteria are two examples.

Vulnerability disclosure

Responsible disclosure of vulnerabilities is a topic of great debate. As reported by The Tech Herald in August 2010, "Google, Microsoft, TippingPoint, and Rapid7 have recently issued guidelines and statements addressing how they will deal with disclosure going forward." ^[27]

A responsible disclosure first alerts the affected vendors confidentially before alerting CERT two weeks later, which grants the vendors another 45 day grace period before publishing a security advisory. ^[28]

A full disclosure is done when all the details of vulnerability is publicized, perhaps with the intent to put pressure on the software or procedure authors to find a fix urgently.

Well respected authors have published books on vulnerabilities and how to exploit them: Hacking: The Art of Exploitation Second Edition is a good example.

Security researchers catering to the needs of the cyberwarfare or cybercrime industry have stated that this approach does not provide them with adequate income for their efforts. ^[29] Instead, they offer their exploits privately to enable Zero day attacks.

The never ending effort to find new vulnerabilities and to fix them is called Computer insecurity.

Vulnerability inventory

Mitre Corporation maintains a list of disclosed vulnerabilities in a system called Common Vulnerabilities and Exposures, where vulnerability are classified (scored) using Common Vulnerability Scoring System (CVSS).

OWASP collects a list of potential vulnerabilities in order to prevent system designers and programmers insert vulnerabilities in the software ^[30]

Vulnerability disclosure date

The time of disclosure of a vulnerability is defined differently in the security community and industry. It is most commonly referred to as "a kind of public disclosure of security information by a certain party". Usually, vulnerability information is discussed on a mailing list or published on a security web site and results in a security advisory afterward.

The **time of disclosure** is the first date a security vulnerability is described on a channel where the disclosed information on the vulnerability has to fulfill the following requirement:

- The information is freely available to the public
- The vulnerability information is published by a trusted and independent channel/source
- The vulnerability has undergone analysis by experts such that risk rating information is included upon disclosure

Identifying and removing vulnerabilities

Many software tools exist that can aid in the discovery (and sometimes removal) of vulnerabilities in a computer system. Though these tools can provide an auditor with a good overview of possible vulnerabilities present, they can not replace human judgment. Relying solely on scanners will yield false positives and a limited-scope view of the problems present in the system.

Vulnerabilities have been found in every major operating system including Windows, Mac OS, various forms of Unix and Linux, OpenVMS, and others. The only way to reduce the chance of a vulnerability being used against a system is through constant vigilance, including careful system maintenance (e.g. applying software patches), best practices in deployment (e.g. the use of firewalls and access controls) and auditing (both during development and throughout the deployment lifecycle).

Examples of vulnerabilities

Vulnerabilities are related to:

- physical environment of the system
- the personnel
- management
- administration procedures and security measures within the organization
- business operation and service delivery
- hardware
- software
- communication equipment and facilities
- and their combinations.

It is evident that a pure technical approach cannot even protect physical assets: you should have administrative procedure to let maintenance personnel to enter the facilities and people with adequate knowledge of the procedures, motivated to follow it with proper care. see Social engineering (security).

Four examples of vulnerability exploits:

- an attacker finds and uses an overflow weakness to install malware to export sensitive data;

- an attacker convinces a user to open an email message with attached malware;
- an insider copies a hardened, encrypted program onto a thumb drive and cracks it at home;
- a flood damage your computer systems installed at ground floor.

Software vulnerabilities

Common types of software flaws that lead to vulnerabilities include:

- Memory safety violations, such as:
 - Buffer overflows
 - Dangling pointers
- Input validation errors, such as:
 - Format string attacks
 - Improperly handling shell metacharacters so they are interpreted
 - SQL injection
 - Code injection
 - E-mail injection
 - Directory traversal
 - Cross-site scripting in web applications
 - HTTP header injection
 - HTTP response splitting
- Race conditions, such as:
 - Time-of-check-to-time-of-use bugs
 - Symlink races
- Privilege-confusion bugs, such as:
 - Cross-site request forgery in web applications
 - Clickjacking
 - FTP bounce attack
- Privilege escalation
- User interface failures, such as:
 - Warning fatigue [31] or user conditioning [32]
 - Blaming the Victim Prompting a user to make a security decision without giving the user enough information to answer it [33]
 - Race Conditions [34] [35]

Some set of coding guidelines have been developed and a large number of static code analysers has been used to verify that the code follows the guidelines.

References

- [1] "The Three Tenets of Cyber Security" (<http://www.spi.dod.mil/tenets.htm>). U.S. Air Force Software Protection Initiative. . Retrieved 2009-12-15.
- [2] Foreman, P: *Vulnerability Management*, page 1. Taylor & Francis Group, 2010. ISBN 978-1-4398-0150-5
- [3] ISO/IEC, "Information technology -- Security techniques-Information security risk management" ISO/IEC FIDIS 27005:2008
- [4] British Standard Institute, Information technology -- Security techniques -- Management of information and communications technology security -- Part 1: Concepts and models for information and communications technology security management BS ISO/IEC 13335-1-2004
- [5] Internet Engineering Task Force RFC 2828 Internet Security Glossary
- [6] CNSS Instruction No. 4009 (http://www.cnss.gov/Assets/pdf/cnssi_4009.pdf) dated 26 April 2010
- [7] a wiki project (<http://fismapedia.org/index.php>) devoted to FISMA
- [8] FISMApedia Vulnerability term (<http://fismapedia.org/index.php?title=Term:Vulnerability>)
- [9] NIST SP 800-30 Risk Management Guide for Information Technology Systems (<http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>)
- [10] Risk Management Glossary Vulnerability (<http://www.enisa.europa.eu/act/rm/cr/risk-management-inventory/glossary#G52>)
- [11] Technical Standard Risk Taxonomy ISBN 1-931624-77-1 Document Number: C081 Published by The Open Group, January 2009.
- [12] "An Introduction to Factor Analysis of Information Risk (FAIR)", Risk Management Insight LLC, November 2006 (http://www.riskmanagementinsight.com/media/docs/FAIR_introduction.pdf);
- [13] Matt Bishop and Dave Bailey. A Critical Analysis of Vulnerability Taxonomies. Technical Report CSE-96-11, Department of Computer Science at the University of California at Davis, September 1996
- [14] Schou, Corey (1996). Handbook of INFOSEC Terms, Version 2.0. CD-ROM (Idaho State University & Information Systems Security Organization)
- [15] NIATEC Glossary (<http://niatec.info/Glossary.aspx?term=6018&alpha=V>)
- [16] Wright, Joe; Jim Harmening (2009) "15" *Computer and Information Security Handbook* Morgan Kaufmann Publications Elsevier Inc p. 257 ISBN 978-0-12-374354-1
- [17] ISACA THE RISK IT FRAMEWORK (registration required) (<http://www.isaca.org/Knowledge-Center/Research/Documents/RiskIT-FW-18Nov09-Research.pdf>)
- [18] Kakareka, Almantas (2009) "23" *Computer and Information Security Handbook* Morgan Kaufmann Publications Elsevier Inc p. 393 ISBN 978-0-12-374354-1
- [19] Technical Report CSD-TR-97-026 Ivan Krsul The COAST Laboratory Department of Computer Sciences, Purdue University, April 15, 1997 (<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.5435&rep=rep1&type=pdf>)
- [20] http://www.ranum.com/security/computer_security/editorials/dumb/
- [21] The Web Application Security Consortium Project, Web Application Security Statistics 2009 (<http://projects.webappsec.org/w/page/13246989/Web-Application-Security-Statistics#APPENDIX2ADDITIONALVULNERABILITYCLASSIFICATION>)
- [22] Ross Anderson. Why Cryptosystems Fail. Technical report, University Computer Laboratory, Cambridge, January 1994.
- [23] Neil Schlager. When Technology Fails: Significant Technological Disasters, Accidents, and Failures of the Twentieth Century. Gale Research Inc., 1994.
- [24] Hacking: The Art of Exploitation Second Edition
- [25] Kiountouzis, E. A.; Kokolakis, S. A. *Information systems security: facing the information society of the 21st century* London: Chapman & Hall, Ltd ISBN 0-412-78120-4
- [26] Bavis, Sanjay (2009) "22" *Computer and Information Security Handbook* Morgan Kaufmann Publications Elsevier Inc p. 375 ISBN 978-0-12-374354-1
- [27] The Tech Herald: The new era of vulnerability disclosure - a brief chat with [[HD Moore (<http://www.thetechherald.com/article.php/201033/6025/The-new-era-of-vulnerability-disclosure-a-brief-chat-with-HD-Moore>)]]
- [28] [[Rapid7 (<http://www.rapid7.com/disclosure.jsp>)] Vulnerability Disclosure Policy]
- [29] Blog post about DLL hijacking vulnerability disclosure (<http://blog.rapid7.com/?p=5325>)
- [30] OWASP vulnerability categorization (<http://www.owasp.org/index.php/Category:Vulnerability>)
- [31] <http://www.freedom-to-tinker.com/?p=459>
- [32] <http://www.cs.auckland.ac.nz/~pgut001/pubs/phishing.pdf>
- [33] <http://blog.mozilla.com/rob-sayre/2007/09/28/blaming-the-victim/>
- [34] <http://www.squarefree.com/2004/07/01/race-conditions-in-security-dialogs/>
- [35] <http://lcamtuf.blogspot.com/2010/08/on-designing-uis-for-non-robots.html>

External links

- Security advisories links from the Open Directory http://www.dmoz.org/Computers/Security/Advisories_and_Patches/
- Languages Standard's group (<http://www.aitcnet.org/isai/>): Guidance for Avoiding Vulnerabilities through Language Selection and Use
- Microsoft Security Response Center (<http://www.microsoft.com/technet/archive/community/columns/security/essays/vulnrbl.msp>): Definition of a Security Vulnerability
- NIST Software Assurance Metrics and Tool Evaluation (SAMATE) project (<http://samate.nist.gov/>)
- Open Source Vulnerability Database (OSVDB) homepage (<http://www.osvdb.org/>)
- Open Web Application Security Project (<http://www.owasp.org/index.php/Category:Vulnerability>)
- Common Vulnerabilities and Exposures (CVE) (<http://www.cve.mitre.org/>)

Web testing

Web testing is the name given to software testing that focuses on web applications. Complete testing of a web-based system before going live can help address issues before the system is revealed to the public. Issues such as the security of the web application, the basic functionality of the site, its accessibility to handicapped users and fully able users, as well as readiness for expected traffic and number of users and the ability to survive a massive spike in user traffic, both of which are related to load testing.

Web Application Performance Tool

A Web Application Performance Tool, also known as (WAPT) is used to test web applications and web related interfaces. These tools are used for performance, load and stress testing of web applications, web sites, web servers and other web interfaces. WAPT tends to simulate virtual users which will repeat either recorded URLs or specified URL and allows the users to specify number of times or iterations that the virtual users will have to repeat the recorded URLs. By doing so, the tool is useful to check for bottleneck and performance leakage in the website or web application being tested.

A WAPT faces various challenges during testing and should be able to conduct tests for:

- Browser compatibility
- Operating System compatibility
- Windows application compatibility where required (especially for backend testing)

WAPT allows a user to specify how virtual users are involved in the testing environment. ie either increasing users or constant users or periodic users load. Increasing user load, step by step is called RAMP where virtual users are increased from 0 to hundreds. Constant user load maintains specified user load at all time. Periodic user load tends to increase and decrease the user load from time to time.

Web security testing

Web security testing tells us whether Web based applications requirements are met when they are subjected to malicious input data.^[1]

- Web Application Security Testing Plug-in Collection for FireFox: <https://addons.mozilla.org/en-US/firefox/collection/webappsec>

Testing the user interface of web applications

Some frameworks give a toolbox for testing Web applications.

Open Source web testing tools

- JMeter: <http://jakarta.apache.org/jmeter/>- Java desktop application for load testing and performance measurement.
- HTTP Test Tool: <http://htt.sourceforge.net/>- Scriptable protocol test tool for HTTP protocol based products.

Windows-based web testing tools

- Quick test Professional - Automated functional and regression testing software from HP.
- LoadRunner - Automated performance and load testing software from HP.
- Rational
- SilkTest - Automation tool for testing the functionality of enterprise applications
- Testing Anywhere - Automation testing tool for all types of testing from Automation Anywhere

References

- [1] Hope, Paco; Walther, Ben (2008), *Web Security Testing Cookbook*, O'Reilly Media, Inc., ISBN 978-0-596-51483-9

Further reading

- Hung Nguyen, Robert Johnson, Michael Hackett: *Testing Applications on the Web (2nd Edition): Test Planning for Mobile and Internet-Based Systems* ISBN 0-471-20100-6
- James A. Whittaker: *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley Professional, February 2, 2006. ISBN 0-321-36944-0
- Lydia Ash: *The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests*, Wiley, May 2, 2003. ISBN 0471430218
- S. Sampath, R. Bryce, Gokulanand Viswanath, Vani Kandimalla, A. Gunes Koru. Prioritizing User-Session-Based Test Cases for Web Applications Testing. Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST), Lillehammer, Norway, April 2008.
- "An Empirical Approach to Testing Web Applications Across Diverse Client Platform Configurations" by Cyntrica Eaton and Atif M. Memon. International Journal on Web Engineering and Technology (IJWET), Special Issue on Empirical Studies in Web Engineering, vol. 3, no. 3, 2007, pp. 227–253, Inderscience Publishers.

White-box testing

White-box testing (also known as **clear box testing**, **glass box testing**, **transparent box testing**, and **structural testing**) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are required and used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. This is analogous to testing nodes in a circuit, e.g. in-circuit testing (ICT).

While white-box testing can be applied at the unit, integration and system levels of the software testing process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

White-box test design techniques include:

- Control flow testing
- Data flow testing
- Branch testing
- Path testing

Compare with black-box testing.

Hacking

In penetration testing, white-box testing refers to a methodology where an ethical hacker has full knowledge of the system being attacked. The goal of a white-box penetration test is to simulate a malicious insider who has some knowledge and possibly basic credentials to the target system.

External links


- BCS SIGIST (British Computer Society Specialist Interest Group in Software Testing): *Standard for Software Component Testing* ^[1], Working Draft 3.4, 27. April 2001.
- <http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf> has more information on control flow testing and data flow testing.
- <http://research.microsoft.com/en-us/projects/pex/Pex> - Automated white-box testing for .NET

References

[1] <http://www.testingstandards.co.uk/Component%20Testing.pdf>

Windmill (testing framework)

Windmill

	
Development status	Active
Operating system	Cross-platform
Type	Software testing framework for Web applications
License	Apache License 2.0
Website	http://www.getwindmill.com/

Windmill is a cross-platform, cross-browser software testing framework, primarily used for testing Web applications.

Most users write tests in Python or in JavaScript, but there is also a library that provides Ruby support. Windmill also provides a recorder tool that allows writing tests without learning a programming language.

Windmill supports all major modern Web browsers, and runs on Microsoft Windows, Mac OS X, and Linux.

X-Machine Testing

The **(Stream) X-Machine Testing Methodology** is a *complete* functional testing approach to software- and hardware testing^[1] that exploits the scalability of the Stream X-Machine model of computation.^[2] Using this methodology, it is likely to identify a finite test-set that exhaustively determines whether the tested system's implementation matches its specification. This goal is achieved by a divide-and-conquer approach, in which the design is decomposed by refinement^[3] into a collection of Stream X-Machines, which are implemented as separate modules, then tested bottom-up. At each integration stage, the testing method guarantees that the tested components are correctly integrated.^[4]

The methodology overcomes formal undecidability limitations by requiring that certain design for test principles are followed during specification and implementation. The resulting scalability means that practical software^[5] and hardware^[6] systems consisting of hundreds of thousands of states and millions of transitions have been tested successfully.

Motivation

Much software testing is merely hopeful, seeking to exercise the software system in various ways to see whether any faults can be detected. Testing may indeed reveal some faults, but can never guarantee that the system is correct, once testing is over. Functional testing methods seek to improve on this situation, by developing a formal specification describing the intended behaviour of the system, against which the implementation is later tested (a kind of conformance testing). The specification can be validated against the user-requirements and later proven to be consistent and complete by mathematical reasoning (eliminating any logical design flaws). Complete functional testing methods exploit the specification systematically, generating test-sets which exercise the implemented software system *exhaustively*, to determine whether it conforms to the specification. In particular:

- Full positive testing: confirms that all desired behaviour is present in the system;
- Full negative testing: confirms that no unintended behaviour is present in the system.

This level of testing can be difficult to achieve, since software systems are extremely complex, with hundreds of thousands of states and millions of transitions. What is needed is a way of breaking down the specification and testing problem into parts which can be addressed separately.

Scalable, Abstract Specifications

Mike Holcombe first proposed using Samuel Eilenberg's theoretical X-machine model as the basis for software specification in the late 1980s.^[7] This is because the model cleanly separates the *control flow* of a system from the *processing* carried out by the system. At a given level of abstraction, the system can be viewed as a simple finite state machine consisting of a few states and transitions. The more complex processing is delegated to the *processing functions* on the transitions, which modify the underlying fundamental data type X . Later, each processing function may be separately exposed and characterized by another X-machine, modelling the behaviour of that system operation.

This supports a divide-and-conquer approach, in which the overall system architecture is specified first, then each major system operation is specified next, followed by subroutines, and so forth. At each step, the level of complexity is manageable, because of the independence of each layer. In particular, it is easy for software engineers to validate the simple finite state machines against user requirements.

Incrementally Testable Specifications

Gilbert Laycock first proposed a particular kind of X-machine, the Stream X-Machine, as the basis for the testing method.^[2] The advantage of this variant was the way in which testing could be controlled. In a Stream X-Machine, the fundamental data type has a particular form: $X = Out^* \times Mem \times In^*$, where In^* is an input stream, Out^* is an output stream, and Mem is the internal memory. The transitions of a Stream X-Machine are labelled with processing functions of the form $\varphi: Mem \times In \rightarrow Out \times Mem$, that is, they consume one input from the input stream, possibly modify memory, and produce one output on the output stream (see the associated article for more details).

The benefits for testing are that software systems designed in this way are *observable* at each step. For each input, the machine takes one step, producing an output, such that input/output pairs may be matched exactly. This contrasts with other approaches in which the system *runs to completion* (taking multiple steps) before any observation is made. Furthermore, layered Stream X-Machines offer a convenient abstraction. At each level, the tester may *forget* about the details of the processing functions and consider the (sub-)system just as a simple finite state machine. Powerful methods exist for testing systems that conform to finite state specifications, such as Chow's W-method.^[8]

Specification Method

When following the (Stream) X-Machine methodology, the first stage is to identify the various types of data to be processed. For example, a word processor will use basic types *Character* (keyboard input), *Position* (mouse cursor position) and *Command* (mouse or menu command). There may be other constructed types, such as $Text ::= Character^*$ (a sequence of characters), $Selection ::= Position \times Position$ (the start and end of the selection) and $Document ::= Text \times Selection \times Boolean$ (the text, a possible selection, and a flag to signal if the document has been modified).

High-Level Specification

The top-level specification is a Stream X-Machine describing the main user interaction with the system. For example, the word processor will exist in a number of states, in which keystrokes and commands will have different effects. Suppose that this word processor exists in the states {**Writing**, **Selecting**, **Filing**, **Editing**}. We expect the word processor to start in the initial **Writing** state, but to move to the **Selecting** state if either the mouse is *dragged*, or the *shift-key* is held down. Once the selection is established, it should return to the **Writing** state. Likewise, if a menu option is chosen, this should enter the **Editing** or **Filing** state. In these states, certain keystrokes may have different meanings. The word processor eventually returns to the **Writing** state, when any menu command has finished. This state machine is designed and labelled informally with the various actions that cause it to change state.

The input, memory and output types for the top-level machine are now formalised. Suppose that the memory type of the simple word processor is the type *Document* defined above. This treats a document as a text string, with two positions marking a possible selection and a flag to indicate modification since the last *save-command*. A more complex word processor might support undoable editing, with a sequence of document states: $Document ::= (Text \times Selection)^*$, which are collapsed to one document every time a *save-command* is performed.

Suppose that the input type for the machine is: $In ::= Command \times Character \times Position$. This recognises that every interaction could be a simple character insertion, a menu command or a cursor placement. Any given interaction is a 3-tuple, but some places may be empty. For example, $(Insert, 'a', \epsilon)$ would represent typing the character 'a'; while $(Position, \epsilon, 32)$ would mean placing the cursor between characters 32 and 33; and $(Select, \epsilon, 32)$ would mean selecting the text between the current cursor position and the place between characters 32 and 33.

The output type for the machine is designed so that it is possible to determine from the output *which* processing function was executed, in response to a given input. This relates to the property of *output distinguishability*, described below.

Low-Level Specification

If a system is complex, then it will most likely be decomposed into several Stream X-Machines. The most common kind of refinement is to take each of the major processing functions (which were the labels on the high-level machine) and treat these as separate Stream X-Machines.^[3] In this case, the input, memory and output types for the low-level machines will be different from those defined for the high-level machine. Either, this is treated as an expansion of the data sets used at the high level, or there is a translation from more abstract data sets at the high level into more detailed data sets at the lower level. For example, a command *Select* at the high level could be decomposed into three events: *MouseDown*, *MouseMove*, *MouseUp* at the lower level.

Ipate and Holcombe mention several kinds of refinement, including *functional refinement*, in which the behaviour of the processing functions is elaborated in more detail, and *state refinement*, in which a simple state-space is partitioned into a more complex state-space.^[1] Ipate proves these two kinds of refinement to be eventually equivalent^[9]

Systems are otherwise specified down to the level at which the designer is prepared to trust the primitive operations supported by the implementation environment. It is also possible to test small units exhaustively by other testing methods.

Design-For-Test Conditions

The (Stream) X-Machine methodology requires the designer to observe certain design for test conditions. These are typically not too difficult to satisfy. For each Stream X-Machine in the specification, we must obtain:

- **Minimal Specification:** The specification must be a *minimal* finite state machine. This means that the state machine should not contain redundant states, that is, states in which the observable transition behaviour is identical to that in some other state.
- **Deterministic Specification:** For each state of the machine, at most one of the processing functions φ should be enabled for the current memory and next input value. This ensures that the required behaviour to be tested is predictable.
- **Test Completeness:** Each processing function φ must be executable for at least one input, with respect to all memory states. This ensures that there are no deadlocks, where the machine is blocked by the current state of memory. To ensure test completeness, the domain of a function φ may be extended with special *test inputs* that are only used during testing.
- **Output Distinguishability:** It must be possible to distinguish which processing function was invoked from its output value alone, for all memory-input pairs. This ensures that the state machine can be decoupled from the processing functions. To ensure output distinguishability, the codomain of a function φ may be extended with special *test outputs* that are only relevant during testing.

A minimal machine is the machine with the fewest states and transitions for some given behaviour. Keeping the specification minimal simply ensures that the test sets are as small as possible. A deterministic machine is required for systems that are predictable. Otherwise, an implementation could make an arbitrary choice regarding which transition was taken. Some recent work has relaxed this assumption to allow testing of non-deterministic machines.^[10]

Test completeness is needed to ensure that the implementation is testable within tractable time. For example, if a system has distant, or hard-to-reach states that are only entered after memory has reached a certain limiting value, then special test inputs should be added to allow memory to be bypassed, forcing the state machine into the distant state. This allows all (abstract) states to be covered quickly during testing. Output distinguishability is the key property supporting the scalable testing method. It allows the tester to treat the processing functions φ as simple labels, whose detailed behaviour may be safely ignored, while testing the state machine of the next integration layer. The unique outputs are witness values, which guarantee that a particular function was invoked.

Testing Method

The (Stream) X-Machine Testing Method assumes that both the design and the implementation can be considered as (a collection of) Stream X-Machines. For each pair of corresponding machines (*Spec*, *Imp*), the purpose of testing is to determine whether the behaviour of *Imp*, the machine of the implementation, exactly matches the behaviour of *Spec*, the machine of the specification. Note that *Imp* need not be a minimal machine - it may have more states and transitions than *Spec* and still behave in an identical way.

To test all behaviours, it must be possible to drive a machine into all of its states, then attempt all possible transitions (those which should succeed, and those which should be blocked) to achieve full *positive* and *negative* testing (see above). For transitions which succeed, the destination state must also be verified. Note that if *Spec* and *Imp* have the same number of states, the above describes the smallest test-set that achieves the objective. If *Imp* has more states and transitions than *Spec*, longer test sequences are needed to guarantee that *redundant* states in *Imp* also behave as expected.

Testing all States

The basis for the test generation strategy is Tsun S. Chow's W-Method for testing finite state automata,^[8] chosen because it supports the testing of redundant implementations. Chow's method assumes simple finite state machines with observable inputs and outputs, but no directly observable states. To map onto Chow's formalism, the functions φ_i on the transitions of the Stream X-Machines are treated simply as labels (inputs, in Chow's terms) and the distinguishing outputs are used directly. (Later, a mapping from real inputs and memory (*mem*, *in*) is chosen to trigger each function φ , according to its domain).

To identify specific states in *Imp*, Chow chooses a *characterization set*, W , the smallest set of test sequences that uniquely characterizes each state in *Spec*. That is, when starting in a given state, exercising the sequences in W should yield at least one observable difference, compared to starting in any other state.

To reach each state expected in *Spec*, the tester constructs the *state cover*, C , the smallest set of test sequences that reaches every state. This can be constructed by automatic breadth-first exploration of *Spec*. The test-set which validates all the states of a minimal *Imp* is then: $C \otimes W$, where \otimes denotes the *concatenated product* of the two sets. For example, if $C = \{<a>, \}$ and $W = \{<c>, <d>\}$, then $C \otimes W = \{<ac>, <ad>, <bc>, <bd>\}$.

Testing all Transitions

The above test-set determines whether a minimal *Imp* has the same states as *Spec*. To determine whether a minimal *Imp* also has the same transition behaviour as *Spec*, the tester constructs the *transition cover*, K . This is the smallest set of test sequences that reaches every state and then attempts every possible transition once, from that state. Now, the input alphabet consists of (the labels of) every function φ in Φ . Let us construct a set of length-1 test sequences, consisting of single functions chosen from Φ , and call this Φ_1 . The transition cover is defined as $K ::= C \cup C \otimes \Phi_1$.

This will attempt every possible transition from every state. For those which succeed, we must validate the destination states. So, the smallest test-set T_1 which completely validates the behaviour of a minimal *Imp* is given by: $T_1 ::= C \otimes W \cup C \otimes \Phi_1 \otimes W$. This formula can be rearranged as:

$$T_1 ::= C \otimes (\Phi_0 \cup \Phi_1) \otimes W,$$

where Φ_0 is the set containing the empty sequence $\{<>\}$.

If *Imp* has more states than *Spec*, the above test-set may not be sufficient to guarantee the conformant behaviour of replicated states in *Imp*. So, sets of longer test sequences are chosen, consisting of all pairs of functions Φ_2 , all triples of functions Φ_3 up to some limit Φ_k , when the tester is satisfied that *Imp* cannot contain chains of duplicated states longer than $k-1$. The final test formula is given by:

$$T_k ::= C \otimes (\Phi_0 \cup \Phi_1 \dots \cup \Phi_k) \otimes W.$$

This test-set completely validates the behaviour of a non-minimal *Imp* in which chains of duplicated states are expected to be no longer than $k-1$. For most practical purposes, testing up to $k=2$, or $k=3$ is quite exhaustive, revealing all state-related faults in really poor implementations.

References

- [1] M. Holcombe and F. Ipate (1998) *Correct Systems - Building a Business Process Solution*. Springer, Applied Computing Series.
- [2] Gilbert Laycock (1993) *The Theory and Practice of Specification Based Software Testing*. PhD Thesis, University of Sheffield. Abstract (<http://www.mcs.le.ac.uk/people/gtl1/PhDabstract.html>)
- [3] F. Ipate and M. Holcombe (1998) 'A method for refining and testing generalised machine specifications'. *Int. J. Comp. Math.* **68**, pp. 197-219.
- [4] F. Ipate and M. Holcombe (1997) 'An integration testing method that is proved to find all faults', *International Journal of Computer Mathematics* **63**, pp. 159-178.
- [5] K. Bogdanov and M. Holcombe (1998) 'Automated test set generation for Statecharts', in: D. Hutter, W Stephan, P. Traverso and M. Ullmann eds. *Applied Formal Methods: FM-Trends 98*, Boppard, Germany, *Lecture Notes in Computer Science* **1641**, pp. 107-121.
- [6] Salim Vanak (2001), *Complete Functional Testing of Hardware Designs*, PhD Thesis, University of Sheffield.
- [7] M. Holcombe (1988) 'X-machines as a basis for dynamic system specification', *Software Engineering Journal* **3**(2), pp. 69-76.
- [8] T. S. Chow (1978) 'Testing software design modelled by finite state machines', *IEEE Transactions on Software Engineering*, **4** (3), pp. 178-187.
- [9] Florentin Ipate (1995) *Theory of X-Machines with Applications in Specification and Testing*, PhD Thesis, Department of Computer Science, University of Sheffield.
- [10] F. Ipate and M. Holcombe (2000) 'Testing non-deterministic X-machines'. In: *Words, Sequences, Grammars, Languages: Where Biology, Computer Science, Linguistics and Mathematics Meet, Vol II*, eds. C Martin-Vide and V. Mitrana, Kluwer.

Article Sources and Contributors

Program animation *Source:* <http://en.wikipedia.org/w/index.php?oldid=442204641> *Contributors:* Abdull, Kdakin, Mission Fleg, The Thing That Should Not Be, 20 anonymous edits

Software testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=446312268> *Contributors:* 0612, 144.132.75.xxx, 152.98.195.xxx, 166.46.99.xxx, 192.193.196.xxx, 212.153.190.xxx, 28bytes, 2D, 2mcm, 62.163.16.xxx, A Man In Black, A R King, A.R., A5b, Abdull, AbsolutDan, Academic Challenger, Acather96, Ad88110, Adam Hauner, Addihockey10, Ag2402, Agopinath, Ahoerstemeier, Ahyl1, Aitias, Akamad, Akhiladi007, AlMac, Alappuzhakaran, Albanaco, Alheny2006, AliaksandrAA, AliveFreeHappy, Allan McInnes, Allstarecho, Alvestrand, Amire80, Amty4all, Andonic, Andre Engels, Andreas Kaufmann, Andres, Andrewmccardle, Andygreeny, Ankurj, Anna Frodesiak, Anna88banana, Annepetersen, Anonymous Dissident, Anonymous anonymous, Anonymous editor, Anorthup, Anthonares, Anwar saadat, Aphstein, Apparition11, Aravindan Shanmugasundaram, ArmadilloFromHell, Ash, Ashdurbat, Avoided, Barunbiswas, Bavinothkumar, Baxtersmalls, Bazzargh, Beland, Bentogoa, Betterusername, Bex84, Bigtwilkins, Bigwyrn, Bilbo1507, Bindu Laxminarayan, Bkll, Blair Bonnett, Blake8086, Bluerasberry, Bobdanny, Bobisthebest, Bobo192, Bonadea, Bornhj, Bovineone, Boxplot, Bpluss, Breno, Brequinda, Brion VIBBER, BruceRuxton, Brunodeschenes.qc, Bryan Derksen, Bsdlogical, Burakseren, Buxbaum666, Calton, CanisRufus, Canterbury Tail, Canterj, CardinalDan, CattleGirl, CemKaner, Certellus, Certes, Cgvak, Chairboy, Chaiths, Chaser, ChiLiBeserker, Chowbok, Chris Pickett, ChrisB, ChrisSteinbach, ChristianEdwardGruber, Chrzastek, Cjhawk22, Claygate, Closedmouth, Cometstyles, Conan, Contributor124, Conversion script, CopperMurdoch, Corruptcopper, Cpl Syx, Cptchipjew, Craigwb, Cvcby, Cybercobra, CyborgTosser, DARTH SIDIOUS 2, DMacks, DRogers, Dacouts, DaisyMLL, Dakart, Dalric, Danhash, Danimal, Davewild, David.alex.lamb, Dazzla, Dbelhumeur02, Dcarrion, Declan Kavanagh, DeltaQuad, Denisarona, Der Falke, DerHexer, Derek farn, Dev1240, Dicklyon, Diego.pamio, Digitalfunda, Discospinster, Dnddnd80, Downsize43, Dravecky, Drewster1829, Drxim, DryCleanOnly, Dvansant, Dvyost, E2eamon, ELinguist, ESKog, Ea8f93wala, Ebde, Ed Poor, Edward Z. Yang, Electiontechnology, ElfriedeDustin, Ellenaz, Enumera, Enviroboy, Epim, Epolk, Eptin, Ericholmstrom, Erkan Yilmaz, ErkinBatu, Esoltas, Excirial, Falcon8765, FalconL, Faught, Felix Wiemann, Flavioxavier, Forlornturtle, FrankCostanza, Fredrik, FreplySpang, Furrykef, G0gogsc300, GABaker, Gail, Gar3t, Gary King, Gary Kirk, Gdavidp, Gdo01, GeoTe, Georgie Canadian, Geosak, Giggy, Gil mo, Gogo Dodo, Goldom, Gonchibolso12, Gorton78, Graemel., GregorB, Gsmgm, Guehene, Gurchzilla, GururajOaksy, Guybrush1979, Hadal, Halovivek, Halstead, HamburgerRadio, Harald Hansen, Havlatm, Haza-w, Hdt83, Headbomb, Helix84, Hemmath18, Henri662, Honey88foru, Hooperbloob, Hsingh77, Hu12, Hubschrauber729, Huge Bananas, Hutch1989r15, I dream of horses, IJA, IceManBrazil, Ignasiokambale, ImALion, Imroy, Incnis Mersi, Indon, Infrogation, Intry, Inwind, J.delanoy, JASpencer, JPFitzmaurice, Ja 62, JacobBramley, Jake Wartenberg, Jeff G., Jehochman, Jenny MacKinnon, JesseHogan, JimD, Jjamon, Jm266, Jmax-, Jmckey, JoeSmack, John S Eden, JohnDI, Johnny.cache, Johnnieq, JonJosephA, Jonesko, JosephDonahue, Josheisenberg, Joshymit, Joyous!, Jsled, Jstastny, Jtowler, Juliancolton, JuneGloom07, Jwooder, Kalkundri, KamikazeArchon, Kanenas, Kdakin, Kevin, Kgf0, Khalid hassani, Kingpin13, Kingpomba, Kitdaddio, Kku, KnowledgeOfSelf, Kompere, Konstable, Krashlandon, Kuru, LeaveSleaves, Lee Daniel Crocker, Leszek Jańczuk, Leujohn, Little Mountain 5, Lomn, Losaltosboy, Lotje, Lowellian, Lradrama, Lumpish Scholar, M Johnson, MER-C, MPerel, Mabdul, Madhero88, Madvin, Mailtoramkumar, Manekari, ManojPhilipMathen, Mark Renier, Materialscientist, MattGiuca, Matthew Stannard, MaxHund, MaxSem, Mazi, Mblumber, Mburdis, Mdd, Mentifisto, Menzogna, Metagraph, Mfactor, Mhatham.shammaa, Michael B. Trausch, Michael Bernstein, MichaelBolton, Michig, Mike Doughney, MikeDogma, Miker@sundialservices.com, Mikethegreen, Misza13, Mitch Ames, Miterdale, Mimgreiner, Moa3333, Mpiilaeten, Mpradeep, Mr Minchin, MrJones, MrOllie, Mrh30, Msm, Mtoxcv, Munaz, Mxn, N8mills, NAHID, Nambika.marian, Nanobug, Neokamek, Newbie59, Nibblus, Nick Hickman, Nigholith, Nimowy, Nksp07, Noah Salzman, Notinasnoid, Novice7, Nuno Tavares, Oashi, Ocee, Oddity-, Ohnoitsjamie, Oicumayberight, Oliver1234, Omicronperseis8, Orange Suede Sofa, Orphan Wiki, Ospalh, Otis80hobson, Oysterguitarist, PL290, Panonomia, Pascal.Tesson, Pashute, Paudelp, Paul August, Paul.h, Pcb21, Peashy, Pepsi12, PhilHibbs, Philip Trueman, PhilipO, PhilippeAntras, Phoe6, Piano non troppo, Plieric, Pinecar, Plainparow, Pmberry, Pointillist, Pmoxis, Poulpy, Ppolpp, Prari, Praveen.karri, Priya4212, Promoa1, Psychade, Puraniksameer, Pysuresh, QTCaptain, Qiassist, Qatutor, Qazwxscdrfvtyghyn, Qwyrxian, RHaworth, Radagast83, Rahuljaitley82, Rajesh mathur, RameshaL.B, Randhirreddy, Ravialluru, Raynald, RedWolf, RekishiEJ, Remi0o, ReneS, Retired username, Rex black, Rgoodermote, Rhobite, Riagu, Rich Farmbrough, Richard Harvey, Ritigalajayasea, Rje, Rjwilmsi, Rlsheehan, Rmattson, Rmstein, Robbie098, Robert Merkel, Robinson wejman, Rockynook, Ronhones, Rone, Roscelese, Rowlye, Rp, Rror, Ruptan, Rwww, S.K., SJP, SP-KP, SURIV, Sachipra, Sachxn, Sam Hooever, Samansouri, Sankshah, Sapphic, Sardanaphalus, Sasquatch525, SatishKumarB, ScaledLizard, ScottSteiner, Scottri, Sega381, Selket, Senatun, Serge Toper, Sergey1984, Shadowcheets, Shanes, Shepmaster, Shimeru, Shimgray, Shishirhegde, Shoejar, Shubo mu, Shze, Silverbullet234, Sitush, Skala7, Skyqa, Slowbro, Smack, Smurrayinchester, Snowolf, Softtest123, Softwaretest1, Softwaretesting1001, Softwaretesting101, Solde, Somdeb Chakraborty, Someguy1221, Sooner Dave, SpaceFlight88, Spadonic, SpigotMap, Spitfire, Srikanth.sharma, Staceyeschneider, Stensult, StaticGull, Stephen Gilbert, Stephenb, Stevezone, Stickee, Storm Rider, Strmore, SunSwOrd, Superbeecat, SwirlBoy39, Sxm20, Sylvainmarquis, T4tarzan, TCL India, Tagro82, Tdjones74021, Techsmith, Teddickey, Tejas81, Terrilja, Testingexpert, Testinggeek, Testmaster2010, ThaddeusB, The Anome, The Thing That Should Not Be, The prophet wizard of the crayon cake, Thehelpfulone, ThomasO1989, ThomasOwens, Thread-union, Thv, Tipeli, Tippers, Tmaufer, Tobias Bergemann, Toddst1, Tommy2010, Tonym88, Trosser, Ttam, Tulkolahten, Tusharpandya, TuttorMouse, Uktim63, Uncle G, Unforgettableid, Useight, Utcursch, Uzma Gamal, VMS Mosaic, Valenciano, Vaniac, Venkatreddy, Venu6132000, Verloren, VernoWhitney, Versageek, Vijay.ram.pm, Vijaythormothe, Vishwas008, Vsoid, W2qasouer, Walter Görlitz, Wifione, Wiki alf, WikiWilliamP, Wikieditor06, Will Beback Auto, Willsmith, Winchelsea, Wlievens, Wombat77, Wwmbs, Yamamoto Ichiro, Yesyoubee, Yngupta, Yosri, Yuckfoo, ZenerV, Zephyrjs, ZhonghuaDragon2, ZooFari, Zurishaddai, 1991 anonymous edits

Portal:Software Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=427545248> *Contributors:* Andreas Kaufmann, Froestel, Gilliam, JDBravo, Jackfork, Mitch Ames, Pinecar, 14 anonymous edits

Accelerated stress testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=443244581> *Contributors:* Bearcat, Katharineamy, NawlinWiki, Zalte1212

Acceptance testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=438053365> *Contributors:* Alphajuliet, Amire80, Amitg47, Apparition11, Ascánder, Bournejc, Caesura, Caltas, CapitalR, Carse, Chris Pickett, Claudio figueiredo, CloudNine, Conversion script, DRogers, DVD R W, Dahcalan, Daniel.r.bell, Davidbatet, Dholm, Divyadeepsharma, Djmckee1, Dlevy-telerik, Eloquence, Emilybache, Enochlau, GTBacchus, GraemeL., Granburguesa, Gwernol, Halovivek, Hooperbloob, Hu12, Hutcher, Hyad, Infrablue, Jamesochter, Jentreadwell, Jgladding, JimJavascrpt, Jmaranz, Jpp, Kaitanen, KSnow, Liftoph, MartinDK, MeijdenB, Meise, Michael Hardy, Midnightcomm, Mifter, Mike Rosoff, Mjemesston, Mjemesston, Mjemesston, Muhandes, Myhister, Newbie59, Normxxx, Old Moonraker, Olson.sr, Panzi, Pearle, PeterBrooks, Phamti, Pill, Pinecar, Qem, RHaworth, RJFerret, Riki, Rlsheehan, Rodasmith, Samuel Tan, Shirulashem, Timmy12, Timo Honkasalo, Toddst1, Viridae, Walter Görlitz, Whaa?, Wikipe-tan, William Avery, Winterst, 150 anonymous edits

Ad hoc testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=443007678> *Contributors:* DRogers, Epim, Erkan Yilmaz, Faught, IQDave, Josh Parris, Ottawa4ever, Pankajkitu, Pinecar, Pmod, Robinson wejman, Solde, Walter Görlitz, 12 anonymous edits

Agile testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442661086> *Contributors:* Agiletesting, Alanby, Although, Chowbok, Eewild, Ehendrickson, Ericholmstrom, GoingBatty, Gurch, Hemmath18, Henri662, Janetgregoryca, Johnnieq, LilHelpa, Luiscolorado, M2Ys4U, Manistar, MathMaven, Mdd, ParaTom, Patrickegan, Pinecar, Pnm, Podge82, Random name, Sardanaphalus, ScottWAmbler, Vaibhav.nimbalkar, Walter Görlitz, Webrew, Weimont, Zonafan39, 60 anonymous edits

All-pairs testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442903292> *Contributors:* Ash, Ashwin palaparthi, Bookworm271, Brandon, Capricorn42, Chris Pickett, Cmdrjameson, Erkan Yilmaz, Kjtobo, LuisCavalheiro, MER-C, Melcombe, MrOllie, Pinecar, Qwfp, Raghu1234, Rajushalem, Regancy42, Rexrange, Rstens, RussBlau, SteveLoughran, Tassedethe, 48 anonymous edits

American Software Testing Qualifications Board *Source:* <http://en.wikipedia.org/w/index.php?oldid=340820953> *Contributors:* Andreas Kaufmann, Ea8f93wala, Malcolmx15, 2 anonymous edits

API Sanity Autotest *Source:* <http://en.wikipedia.org/w/index.php?oldid=433087341> *Contributors:* Andrey86, Eekster, Tognopop, 1 anonymous edits

Association for Software Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=398282105> *Contributors:* CemKaner, Csguy2, Dawynn, Dispenser, Faught, Giraffedata, Kgf0, Pinecar, ReyBrujo, Rsbarrer, Softtest123, 11 anonymous edits

Attack patterns *Source:* <http://en.wikipedia.org/w/index.php?oldid=404146897> *Contributors:* Bachrach44, Bobbyquine, DouglasHeld, Dudecon, Enauspeaker, Falcon Kirtaran, FrankTobia, Friedfish, Hooperbloob, Jkelly, Manionc, Natalie Erin, Nono64, Od Mishehu, R00m c, Retired username, Rich257, RockyH, Smokizky, 3 anonymous edits

Augmented Reality-based testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=425241287> *Contributors:* Aandelkovic, Chevymontecarlo, Mild Bill Hiccup

Australian and New Zealand Testing Board *Source:* <http://en.wikipedia.org/w/index.php?oldid=439723472> *Contributors:* ANZTB, Jimmy Pitt, JosieCutts, Pascal666, 1 anonymous edits

Automated Testing Framework *Source:* <http://en.wikipedia.org/w/index.php?oldid=432504216> *Contributors:* Abdull, AvicAWB, G0gogsc300, Letdorf, Msporled, MuffledThud, 3 anonymous edits

Avalanche (dynamic analysis tool) *Source:* <http://en.wikipedia.org/w/index.php?oldid=372464051> *Contributors:* Iisaev

User:Awright415/Centercode *Source:* <http://en.wikipedia.org/w/index.php?oldid=444320206> *Contributors:* Bearcat, BennyIuo, Ihoicy, LukeFreiler, Pagebased1, Reaper Eternal

Debugging *Source:* <http://en.wikipedia.org/w/index.php?oldid=409176429> *Contributors:* Andreas Kaufmann, Dawynn, Erkan Yilmaz, Foobiker, Jchaw, Kaihsu, O keys, 6 anonymous edits

Behavior Driven Development *Source:* <http://en.wikipedia.org/w/index.php?oldid=446176971> *Contributors:* 16x9, Adamleggett, Alexott, Ashsearle, Aslak.hellesoy, BenAveling, Bill.albing, Brolund, CLW, Candace Gillhoolley, Chaos, Colonies Chris, DanNorth, Davemarshal04, David Monaghan, Dcazzulino, Deanberris, Diego Moya, Doekman, Dols, Eleusis, Erkan Yilmaz, Espo, FGeorges, Featheredwings, GhettoBlaster, Giardante, Greenrd, GregorB, Haakon, Hugobr, Humanmatters, Huttarl, Ianspence, Ignu, JLaTondre, Jania902, Jaxelrod, Jbandi, Johnmarkos, Johnwyles, Jutame, Kelly Martin, KellyCoinGuy, Kevin Rector, KrzysztofKlimonda, Lenin1991, Lennarth, MMSqueira, MaxSem, Mhennemeyer, Michael miceli, Mortense, MrOllie, Ncrause, Oleganza, Paulmarrington, Peter Karlsen, Philippe, Retteatst, Rick Jelliffe, Rjray, Rjwilmsi, Rodrige, Ryanmilmoyl, Secret9, Some jerk on the Internet, Some standardized rigour, StephenFerg, SteveDonie, Steveonjava, Timhaughton, Tomjadams, TonyBallu, Vinunlr, Wesley, Weswilliams, Where next Columbus?, Yamaguchi先生, Yukoba, 162 anonymous edits

Black-box testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442898546> *Contributors:* A bit iffy, AKGhetto, Aervanath, Ag2402, AndreiW, Andrewpmk, Ash, Asparagus, Benito78, Betterusername, Blake-, CWY2190, Caesura, Canterbury Tail, Chris Pickett, Chrys, ClementSeveillac, Colinky, Courcelles, DRogers, DanDoughty, Daveydwieb, Discospinster, DividedByNegativeZero, Dochoat, DylanW, Ebde, Electiontechnology, Epim, Erkan Yilmaz, ErkinBatu, Fluzwup, Frap, Gayathri nambiar, Geeoharee, Haymaker, Hooperbloob, Hu12, Hughglaser, Ian Pitchford, Ileshko, Isnow, Jmabel, Jondel, Karl Naylor, Kgfo, Khym Chanur, Kuru, LOL, Lahiru k, Lambchop, Liao, Mark.murphy, Mathieu, Michael Hardy, Michig, Mpilaeten, Mr Minchin, MrOllie, NEUROO, NawlinWiki, Nitingai, Notinasnaid, OIEnglish, Otheus, PAS, PerformanceTester, Picaroon, Pinecar, Poor Yorick, Pradameinhoff, Radiojon, Retiono Virginian, Rich Farmbrough, Rstens, Rsutherland, Rwww, S.K., Sergei, Shadowjams, Shijaz, Solde, Subversive.sound, SuperMidget, Tedickey, Tobias Bergemann, Toddst1, UnitedStatesian, WJBscribe, Walter Görlitz, Xiaosflux, Zephyrjs, 207 anonymous edits

Block design *Source:* <http://en.wikipedia.org/w/index.php?oldid=433090028> *Contributors:* Btyner, Charles Matthews, Cullinane, Cicero, David Eppstein, Docu, Dominus, Giftlite, Howard McCay, Joriki, Kaganer, Kiefer.Wolfowitz, Kmhmh, Linas, Mairi, Maproom, Mendelso, Michael Hardy, Mjcollins68, Nassrat, Nbarth, Ott2, Qwfp, RFBailey, Rich Farmbrough, Rogério Brito, Silverfish, SlumdogAramis, Ttz, Vicarious, Will Orrick, Xezbeth, Zaslav, 14 anonymous edits

Boundary case *Source:* <http://en.wikipedia.org/w/index.php?oldid=370928380> *Contributors:* Aervanath, Andreas Kaufmann, Antonielly, Charles Matthews, Dougher, Emurphy42, Gurch, Pinecar, Radagast83, Rl, Wwagner, 5 anonymous edits

Boundary testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=394333584> *Contributors:* DanDoughty, Delian31, Qatutor, 3 anonymous edits

Boundary-value analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=440330386> *Contributors:* Ahoerstemeier, Andreas Kaufmann, AndreiW, Attilios, Benito78, Ccady, DRogers, Duggpm, Ebde, Eumolpo, Freek Verkerk, Ian44, IceManBrazil, Jowler, Krishjugal, Linuxbabu, Michaeldunn123, Mirokado, Pinecar, Psiphorg, Radiojon, Retired username, Robinson weijman, Ruchir1102, Sesh, Stemburn, Stemonitis, Sunithasiri, Velella, Walter Görlitz, Wisgary, Zoz, 59 anonymous edits

Browser speed test *Source:* <http://en.wikipedia.org/w/index.php?oldid=437013681> *Contributors:* Aavindraa, Dawynn, Gyrobo, JWilliamCupp, Mabdul, Pointillist, RehmT, Sitush, WOSlinker, 7 anonymous edits

BS 7925-1 *Source:* <http://en.wikipedia.org/w/index.php?oldid=412806925> *Contributors:* ErrantX, Gary King, PamD, Reddana, Sophus Bie

BS 7925-2 *Source:* <http://en.wikipedia.org/w/index.php?oldid=412805579> *Contributors:* ErrantX, Gary King, PamD, Reddana, Sophus Bie

Bug bash *Source:* <http://en.wikipedia.org/w/index.php?oldid=434408515> *Contributors:* Andreas Kaufmann, Archippus, BD2412, Cander0000, DragonflySixtyseven, Freek Verkerk, MisterHand, Pinecar, Retired username, Thumperward, 1 anonymous edits

Build verification test *Source:* <http://en.wikipedia.org/w/index.php?oldid=397932749> *Contributors:* Ackoz, Busy Stubber, Culix, Dawynn, Melaen, Pinecar, SJP, Virendrap, Viridae, Where, 9 anonymous edits

CA/EZTEST *Source:* <http://en.wikipedia.org/w/index.php?oldid=442332434> *Contributors:* Chevymontecarlo alt, Hongooi, Jpbowen, Kdakin, Ospalh, PigFlu Oink, Rror, Rwww, The Epopt, Wayne Miller, 11 anonymous edits

Cause-effect graph *Source:* <http://en.wikipedia.org/w/index.php?oldid=423165981> *Contributors:* Andreas Kaufmann, Bilbo1507, DRogers, Nbarth, OllieFury, Pgr94, Rjwilmsi, The Anome, Weizeero, 2 anonymous edits

Characterization test *Source:* <http://en.wikipedia.org/w/index.php?oldid=445835207> *Contributors:* Alberto Savoia, Andreas Kaufmann, BrianOfRugby, Colonies Chris, David Edgar, Dbenbenn, GabrielSjoberg, JLaTondre, Jamison, Jkl, Mathiasck, PhilippeAntras, Pinecar, Robofish, Swtechwr, Ulner, 12 anonymous edits

Cloud testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=439596324> *Contributors:* Alexei.led, Alongir, D6, Drmies, Feinoha, Gacpro, JHowardGiddings, Johnuniq, Kku, Minimac, Nakon, Neutrality, PhilSmith2, Piano non troppo, Pinecar, PrimeHunter, Ron Ritzman, SamJohnston, Swandove, Tkdoach, Vanruvan, VernoWhitney, WikiScrubber, Zhangw187, 21 anonymous edits

Code coverage *Source:* <http://en.wikipedia.org/w/index.php?oldid=446190361> *Contributors:* 194.237.150.xxx, Abdull, Abednigo, Ad88110, Agasta, Aisingdonnelly, Aitias, Aivosto, AliveFreeHappy, Alksub, Allen Moore, Altenmann, Andreas Kaufmann, Andresmlinar, Anorthup, Attilios, Auteurs, Beetstra, BenFrantzDale, Bingbangbong, BlackMamba, Blacklily, Blaxthos, Chester Markel, Conversion script, CoverageMeter, DagErlingSmørgrav, Damian Yerrick, Derek farn, Digantorama, Dr ecksk, Ebelular, Erkan Yilmaz, Faulknerck2, FredCassidy, Gaudol, GhettoBlaster, Gibber blot, Greensburger, HaeB, Henri662, Hertzsprung, Hob Gadling, Hooperbloob, Hqb, Infofred, JASpencer, JJMax, Jamelan, JavaTenor, Jdpipe, Jerryobject, Jkeen, Johannes Simon, JorisV, Jtheires, Julius.shaw, Kdakin, Kku, Kurykh, LDRA, LouScheffer, M4gnum0n, MER-C, MaterialsScientist, Mati22081979, Matt Crypto, Millerlyte87, Miracleworker5263, Mj1000, MrOllie, MywikiaccountSA, Nat hillary, NawlinWiki, Nigelj, Nin1975, Nixeeagle, Ntalamai, Parasoft-pl, Penumbra2000, Phatom87, Pinecar, Ptrb, Quamrana, Quinntaylor, Quux, RedWolf, Roadbiker53, Robert Merkel, Rpapo, RuggeroB, Rwww, Scubamunki, SebastianDietrich, SimonKagstrom, Smharr4, Snoyes, Suruena, Taibah U, Technoparkcorp, Test-tools, Testcocon, Tiagofassoni, TutterMouse, U2perkunas, Veralift, Walter Görlitz, WimdeValk, Witten rules, Wlievens, Wmwmurray, X746e, 217 anonymous edits

Code integrity *Source:* <http://en.wikipedia.org/w/index.php?oldid=308702814> *Contributors:* David smallfield, Malcolma

Codenomicon *Source:* <http://en.wikipedia.org/w/index.php?oldid=431617952> *Contributors:* Andreas Kaufmann, Ari.takanen, Frap, Gadaloo, Gary King, Hmains, JIP, Joneskoo, Nynex01, Purpleshoe, Rich Farmbrough, RockMFR, T0pgear09, Wizard191, 10 anonymous edits

Compatibility testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=370597747> *Contributors:* Alison9, Arkitus, Iain99, Jimj wpg, Kumar74, Neelov, Pinecar, RekishiEJ, Rwww, 4 anonymous edits

Component-Based Usability Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442206940> *Contributors:* Michael Hardy, Walter Görlitz, Wilhelmina Will, Willem-Paul

Conference Room Pilot *Source:* <http://en.wikipedia.org/w/index.php?oldid=379064418> *Contributors:* D6, Dawynn, Fabrictramp, Interchange88, JPFitzmaurice, Kathleen.wright5, Keesiewonder, Mpelphey, RJFJR, Sellisuk, 7 anonymous edits

Conformance testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=402795298> *Contributors:* Brandon.irwin, Bryan Derksen, Cfeet77, Cmdrjameson, Cutler, Edward, Flameviper, Frap, JonHarder, Kaymai, Mbell, Mindmatrix, Peter Gulutzan, Pinecar, R'n'B, Raviialuru, RexxS, Robert Merkel, Thv, WCrosan, 8 anonymous edits

Core Security *Source:* <http://en.wikipedia.org/w/index.php?oldid=443907248> *Contributors:* Alvin Seville, Bellh2007, Coresec, DanielRigal, Hinesmatt, Katharineamy, Lamro, Pastore Italy, 2 anonymous edits

Corner case *Source:* <http://en.wikipedia.org/w/index.php?oldid=444488077> *Contributors:* Alvis, Calbaer, Chaos5023, Dorftrottel, Dtobias, FlashSheridan, Fluzwup, Gardar Rurak, Helvitica Bold, Hooperbloob, Khym Chanur, LX, Nurg, Pakaran, Pinecar, Radagast83, Retired username, Rickyp, Ripper234, Rl, Sam Hocevar, Sukoshisumo, Tregoweth, Walter Görlitz, 12 anonymous edits

Daikon (system) *Source:* <http://en.wikipedia.org/w/index.php?oldid=413691241> *Contributors:* Bigbluefish, El Pantera, Greenrd, JLaTondre, Katharineamy, 4 anonymous edits

Data-driven testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=407028148> *Contributors:* 2Alen, Amorymeltzer, Andreas Kaufmann, Cornellrockey, EdGl, Fabrictramp, MrOllie, Mrinmayee.p, Phanisrikar, Pinecar, Rajwiki, Rjwilmsi, Rwww, SAE1962, Sbono, Sean.co.za, Zaphodikus, 32 anonymous edits

Decision table *Source:* <http://en.wikipedia.org/w/index.php?oldid=441578241> *Contributors:* Airsplit, Akhvee, Bilbo1507, Canthusus, DNewhall, DRogers, Dulciana, Fast healthy fish, Freek Verkerk, Graham87, Hadal, Hugh glaser, Imcdnzl, JQF, Jacobfeldman, Kuru, Lawranj, Linuxbabu, LizardJr8, Magioladitis, Marek69, Martian, Martinkeesen, Mat cross, MaxHund, Media lib, Michael Hardy, Mstefan, NawlinWiki, Paulasnow, Paulsnx2, Piisa, Pnm, Poccil, RHaworth, Rlsheehan, Sir48, Snori, Thekohser, Toddst1, Ugly bag of mostly water, Useight, Wallnerm, Walter Görlitz, Wavelength, Wiki alf, Wiki-art-name, Wikipelli, 76 anonymous edits

Decision-to-decision path *Source:* <http://en.wikipedia.org/w/index.php?oldid=435590512> *Contributors:* Andreas Kaufmann, Bilbo1507, DI2000, Intgr, 4 anonymous edits

Design predicates *Source:* <http://en.wikipedia.org/w/index.php?oldid=442855063> *Contributors:* Andreas Kaufmann, DRogers, Pinecar, Rich Farmbrough, Tikiwont, 6 anonymous edits

Development, testing, acceptance and production *Source:* <http://en.wikipedia.org/w/index.php?oldid=442191645> *Contributors:* Bearcat, Bert.Roos, Blanchardb, Rjwilmsi, Ties

DeviceAnywhere *Source:* <http://en.wikipedia.org/w/index.php?oldid=432209105> *Contributors:* CommonsDelinker, CyberGhostface, Deviceanywhere, Huunex, JLaTondre, Miami33139, RasterFaAye, Sgroupace, Star Mississippi, Topperfalkon, Woohookitty, 3 anonymous edits

Dry run (testing) *Source:* <http://en.wikipedia.org/w/index.php?oldid=440844976> *Contributors:* AndrewHowse, Busy Stubber, Catrope, Chrisportelli, Crzussian, Farosdaughter, FirefoxRocks, Garion96, Greenrd, Hooperbloob, Horus Kol, Ixfd64, Jeodesic, Kaustuv, Mickthefish, NerdyNSK, Pastordavid, Pinecar, Qxz, Rwww, Snail Doom, Sori4mvp, Takeda, The Thing That Should Not Be, Train2104, Uncle G, Vsatyaramesh, WWB, 24 anonymous edits

Dynamic program analysis *Source:* <http://en.wikipedia.org/w/index.php?oldid=441742709> *Contributors:* A5b, Aivosto, AliveFreeHappy, Antonielly, AvicAWB, Bernard François, D.scain.farenzena, DatabACE, Dbelhumeur02, Derek farn, Dmitry.Bedrin, Dodel, Doramjan, Ehn, Gf uip, lisaev, Jabraham mw, Jayabra17, Jayden54, Jshiss, Kdakin, LDRA, Mmernex, NathanoNL, Pcap, Quarui, RHaworth, Raul654, Rpapo, Ruud Koot, Swtechwr, Syntern, Technobadger, Tracerbee, WikiSlasher, 23 anonymous edits

Dynamic testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=424815549> *Contributors:* 28bytes, Andreas Kaufmann, El Pantera, Gabriel1907, Gary King, Jluedem, Kauczuk, Nla128, Pinecar, Qem, Retired username, Roshan baladharvi, Rossj81, Schmokerash, Sripradha, The Rambling Man, Walter Görlitz, 26 anonymous edits

Edge case *Source:* <http://en.wikipedia.org/w/index.php?oldid=354422617> *Contributors:* Aervanath, ChrisHardie, Hooperbloob, LX, Pinecar, Radagast83, Ripper234, RosinDebow, Sukoshisumo, Ufwuct, Ward3001, 5 anonymous edits

Endeavour Software Project Management *Source:* <http://en.wikipedia.org/w/index.php?oldid=444032525> *Contributors:* CBM, E-cuellar, Gioto, Haakon, Luen, Ninadpachpute, Pol098, Rcawsey, Rich Farmbrough, Rodamaker, Stifle, 16 anonymous edits

Equivalence partitioning *Source:* <http://en.wikipedia.org/w/index.php?oldid=438916426> *Contributors:* Attilios, AvicAWB, Blaisorblade, DRogers, Dougher, Ebde, Erechtheus, Frank1101, HossMo, Ianr44, Ingenhut, JennyRad, Jerry4100, Jj137, Jtowler, Kjtobo, Martinkeesen, Mbrann747, Michig, Mirokado, Pinecar, Rakesh82, Retired username, Robinson weijman, SCEhardt, Stephan Leeds, Sunithasiri, Tedickey, Throw it in the Fire, Vasinov, Walter Görlitz, Wisgray, Zoz, 34 anonymous edits

Error guessing *Source:* <http://en.wikipedia.org/w/index.php?oldid=441193086> *Contributors:* Andreas Kaufmann, Attilios, BD2412, Cooldeal, Debasis.maverick, Fabrictramp, Freek Verkerk, Oferprat, Rickjpellg, Salmar, ThisIsAce, ThomasO1989, Tinucherian, Walter Görlitz, 5 anonymous edits

Exploratory testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=440062272> *Contributors:* Alai, BUPHAGUS55, Bender235, Chris Pickett, DRogers, Decltype, Doab, Dougher, Elopio, Epim, Erkan Yilmaz, Fiftyquid, GoingBatty, IQDave, Imageforward, Jeff.fry, JnRouvignac, Kgf0, Lakeworks, Morrillonline, Mpilaeten, Oashi, Pinecar, Quercus basaseachicensis, Shadowjams, SiriusDG, Softtest123, SudoGhost, TheParanoidOne, Toddst1, Vegaswikian, VilleAine, Walter Görlitz, Whylom, 49 anonymous edits

Fagan inspection *Source:* <http://en.wikipedia.org/w/index.php?oldid=433776670> *Contributors:* Altenmann, Arthena, Ash, Attilios, Bigbluefish, Can't sleep, clown will eat me, ChrisG, Courcelles, Drbreznjev, Epefleche, Gaff, Gaius Cornelius, Gimmetrow, Hockeyc, Icarusgeek, Iwearavolcomhat, JIP, Kezz90, MacGyverMagic, Mjevans, Mkdjadhav, Nick Number, Okok, Pedro.haruo, Slightsmile, Tagishsimon, Talkaboutquality, Tassedethe, The Font, The Letter J, Zundark, 43 anonymous edits

Fault injection *Source:* <http://en.wikipedia.org/w/index.php?oldid=446109779> *Contributors:* Andreas Kaufmann, Ari.takanen, Auntof6, CapitalR, Chowbok, CyborgTosser, DaGizza, DatabACE, Firealwayworks, Foobiker, Jeff G., Joriki, Paff1, Paul.Dan.Marinescu, Piano non troppo, RHaworth, SteveLoughran, Suruena, Tmaufer, Tony1, WillDo, 26 anonymous edits

Financial tester *Source:* <http://en.wikipedia.org/w/index.php?oldid=368966451> *Contributors:* JuJube, Rjwilmsi, Shintekk, Utcursch

Framework for Integrated Test *Source:* <http://en.wikipedia.org/w/index.php?oldid=415741548> *Contributors:* Amire80, Dr ecksk, Elendal, Michig, Moez, Pinecar, Stumps, Tassedethe, Theodoros Bruchwald, ThomasOwens, Torc2, Yrjö Kari-Koskinen, 6 anonymous edits

Functional testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=439237461> *Contributors:* Anonymous monkey, AvicAWB, Chris Pickett, Culix, DRogers, Decrease789, GTBacchus, Henri662, Ontist, Wotan, 20 anonymous edits

Functionality assurance *Source:* <http://en.wikipedia.org/w/index.php?oldid=345202402> *Contributors:* DaGizza, Dawynn, FlyHigh, Fvw, Gaius Cornelius, Malcolm, Melesse, PaulHanson, Pearle, RJFJR, Shell Kinney, 3 anonymous edits

Fuzz testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=445315430> *Contributors:* Andypdavis, Aphstein, Ari.takanen, Autarch, Blashyrk, Bovlb, David Gerard, Dcoetzee, Derek farn, Dirkkb, Doradus, Edward, Emurphy42, Eric Naval, ErrantX, Fluffernutter, FlyingToaster, Furrykef, GregAsche, Guy Harris, Gwern, Haakon, HaeB, Hooperbloob, Hu12, Informationh0b0, Irishguy, Jim.henderson, JonHarder, Jruderman, Kgfleischmann, Kku, Leonard G., Malvineous, Manuel.oriol, Marqued, Martinmeyer, Marudubshinki, McGeddon, Mezzaluna, MikeEddington, Monty845, Mpeisenbr, MrOllie, Nandhp, Neale Monks, Neelix, Niri.M, Pinecar, Posix memalign, Povman, Resprinter123, Ronz, Sadeq, Softtest123, Starofale, Stephanakib, Stevehuges, SwissPokey, T0pgear09, The Anome, The Cuncator, Tmaufer, Tremilux, User At Work, Victor Stinner, Walter Görlitz, Yurymik, Zarkthehackeralliance, Zippy, Zirconscoot, 138 anonymous edits

Game testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=445293822> *Contributors:* A F K When Needed, Adraeus, Anaxial, Andygreeny, Benweedon, Bongwarrior, BradBaker123, Bwilkins, Chintu rohit, Christian75, Citicact, Cpl Sys, Daveydweeb, Effigynmotion, Excirial, Frecklefoot, Geosak, H3llkn0wz, Ham Pastrami, Harry, Hollowtone, Interior, Ixfd64, JLaTondre, JPG-GR, Jappalang, Jmartinson, Katieh5584, Knuckles sonic8, LanceSB, M.thoriyan, Marasmusine, Maxim, Owlpal, PamD, Pinecar, PowerUpGames, Pr4shanthk, Purush97k, RB972, Readro, Rich Farmbrough, ShakingSpirit, SharkD, Some guy, Some jerk on the Internet, Stifle, Subversive.sound, Tester-mike, The Utahraptor, Thearibter, Tiyang, Tommy2010, Ubiq, Veinor, Walter Görlitz, Wikipelli, Xionbox, Yoasif, 133 anonymous edits

Google Guice *Source:* <http://en.wikipedia.org/w/index.php?oldid=437398853> *Contributors:* Arnouldvm, BMorearty, Bearcat, Coldb00t, Cybercobra, Daniel.Cardenas, DudeOfOne, Eleassar, Fabrictramp, GreyTeardrop, HamburgerRadio, Hertzsprung, Honeplus, Jlmayorga, Kks krishna, Klimov, MER-C, Psychohexane, Regniraj, Stormbay, Swinburner, Tainter, ThurnerRupert, Ursus2, William Ortiz, Yacoob, 17 anonymous edits

Graphical user interface testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=443622406> *Contributors:* 10metreh, Alexius08, AliveFreeHappy, Andreas Kaufmann, Chaser, Cmbay, Craigwb, Dreftymac, Dru of Id, Equatin, Gururajs, Hardburn, Hesa, Hu12, Imroy, Jeff G., JnRouvignac, Josepthate, Jruuska, Jwoodger, Ken g6, Liberatus, MER-C, Mcristinel, Mjdjohns5, Mild Bill Hiccup, Paul6feet1, Pinecar, Pnm, Rdancer, Rich Farmbrough, Rjwilmsi, Rockfang, Ronz, SAE1962, SiriusDG, Staceyeschneider, SteveLoughran, Steven Zhang, Unforgettableid, Wahab80, Wakusei, 54 anonymous edits

Hybrid testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=366545628> *Contributors:* Bunnyhop11, Horologium, MrOllie, Vishwas008, 5 anonymous edits

IBM Product Test *Source:* <http://en.wikipedia.org/w/index.php?oldid=438777390> *Contributors:* Bonglander, Dav4is, Dougweller, Frank Lofaro Jr., Nick Number, PiMaster3, RJFJR, Rich Farmbrough, Rwww, 5 anonymous edits

IBM Rational Quality Manager *Source:* <http://en.wikipedia.org/w/index.php?oldid=436026551> *Contributors:* Akerans, Bbryson, Dethlock99, Jezhotwells, Kishanadh, Kku, Welsh, 7 anonymous edits

IEEE 829 *Source:* <http://en.wikipedia.org/w/index.php?oldid=430442333> *Contributors:* A.R., Antariksawan, CesarB, Das.steinchen, Donmillion, Firefox13, Fredrik, GABaker, Grendelkhan, Haakon, Inukjuak, J.delanoy, Korath, Matthew Stannard, Methylgrace, Nasa-serve, Pinecar, Pmberry, Robertvan1, Shizhao, Utuado, Walter Görlitz, 37 anonymous edits

Independent software verification and validation *Source:* <http://en.wikipedia.org/w/index.php?oldid=404709149> *Contributors:* Andreas Kaufmann, Fabrictramp, Firebat08, Mission Fleg, Nikkimaria, Npsiva, Parodrig, RHaworth, Rjwilmsi, Theresa knott, 1 anonymous edits

Installation testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=427188679> *Contributors:* April kathleen, Aranel, Catrope, CultureDrone, Hooperbloob, Matthew Stannard, MichaelDeady, Mr.sqa, Paulbulman, Pinecar, Telestylo, TheParanoidOne, WhatamIdoing, 14 anonymous edits

Integration testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=444050552> *Contributors:* 2002:82ec:b30a:badf:203:baff:fe81:7565, Abdull, Addshore, Amire80, Arunka, Arzach, Cbenedetto, Cellovergara, ChristianEdwardGruber, DRogers, DataSurfer, Discospinster, Ehabmehedi, Faradayplank, Furrykef, Gggg, Gilliam, GreatWhiteNortherner, Hooperbloob, J.delanoy, Jewbacca, Jiang, Jtowler, Kmerenkov, Krashlondon, Lordfaust, Mheusser, Michael Rawdon, Michael miceli, Michig, Myhister, Notinasnaid, Onebyone, Paul August, Pegship, Pinecar, Qaddosh,

Ravedave, Ravindrat, SRCHFD, SkyWalker, Solde, Spokeninsanskrit, Steven Zhang, Svick, TheRanger, Thv, Walter Görlitz, Wyldtwyst, Zhenqinli, 136 anonymous edits

Integration Tree *Source:* <http://en.wikipedia.org/w/index.php?oldid=423307894> *Contributors:* Dawynn, Rwww, SAE1962, Simeon, 1 anonymous edits

International Software Testing Qualifications Board *Source:* <http://en.wikipedia.org/w/index.php?oldid=445740131> *Contributors:* Allan McInnes, Andreas Kaufmann, AssetBurned, Chaitrechait, Czesochowa, Ea8f93wala, Erkan Yilmaz, Issihazan, JASpencer, JonHarder, Jowler, Lexlex, Lmgamaral, Lyzzy, Mandarax, MaterialsScientist, MattiVuori, Niteshch Jain, Pinecar, Thom1306, Walter Görlitz, Whqp, Wizzard, Writeoabhi, 36 anonymous edits

International Software Testing Qualifications Board Certified Tester *Source:* <http://en.wikipedia.org/w/index.php?oldid=433022104> *Contributors:* Beeblebrox, Binksternet, Cander0000, Chrullrich, D6, Derek farn, Digitalfunda, Ea8f93wala, Erkan Yilmaz, Haleyga, JBsupreme, John of Reading, Jowler, Lexlex, Lyzzy, Malcuthrad, MattiVuori, MeilePosthuma, Michig, Mikeoconnell101, Pinecar, Pointer1, Robert Dankanin, Rockynook, Ron Ritzman, Ronz, X2443, 37 anonymous edits

JSystem *Source:* <http://en.wikipedia.org/w/index.php?oldid=426772326> *Contributors:* Andreas Kaufmann, DanielPharos, Fabrictramp, Gderazon, Gil mo, Giraffedata, Katharineamy, Tassedethe, Woohookitty, 3 anonymous edits

Keyword-driven testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=444247690> *Contributors:* 5nizza, Culudamar, Erkan Yilmaz, Heydaysoft, Hooperbloob, Jeff seattle, Jessewgibbs, Jonathan Webley, Jonathon Wright, Jowler, Ken g6, Lowmagnet, Maguschen, MrOllie, Phanisrikar, Pinecar, Rjwilmsi, Rwww, SAE1962, Scraimer, Sean.co.za, Sparrowman980, Swsterinca, Ukkuru, Ultimius, Yun-Yuuzhan (lost password), Zoobeerhall, 57 anonymous edits

Learnability *Source:* <http://en.wikipedia.org/w/index.php?oldid=353691211> *Contributors:* Ivansanchez, Matthew Stannard, Mitch Ames, Per Ardua, Pgr94, 8 anonymous edits

Lightweight software test automation *Source:* <http://en.wikipedia.org/w/index.php?oldid=414865787> *Contributors:* Colonies Chris, Greenrd, JamesDmccaffrey, John Vandenberg, OracleDBGuru, Pnm, Torc2, Tutterz, Verbal, 9 anonymous edits

Load testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=443731698> *Contributors:* AbsolutDan, ArrowmanCoder, BD2412, Bbryson, Belmont, Bernard2, CanadianLinuxUser, Crossdader, Czei, Derby-ridgeback, Dhiraj1984, El Tonerino, Ettrig, Faght, Ff1959, Gail, Gaius Cornelius, Gene Nygaard, Gordon McKeown, Gururajs, Hooperbloob, Hu12, Icairns, Informationh0b0, JHunterJ, JaGa, Jo.witte, Joe knepley, Jpg, Jpo, Jruuska, Ken g6, LinguistAtLarge, M4gnum0n, MER-C, Manzee, Merrill77, Michig, NameIsRon, Nimowy, Nurg, PerformanceTester, Philip2001, Photodeus, Pinecar, Pushtotest, Radagast83, Ravialluru, Rklawton, Rlonn, Rlsheehan, Robert.maclea, Rstens, S.K., Scoops, ScottMasonPrice, Shadowjams, Shadriner, Shashi1212, Shilpagpt, Shinhan, SireenOMari, SpigotMap, Swtechwr, Testgeek, Tusharpandya, Veinor, VernoWhitney, Wahab80, Walter Görlitz, Whitejay251, Wilsonmar, Wrp103, 157 anonymous edits

Localization testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=337792334> *Contributors:* Andreas Kaufmann, Chaitalid, Dawn Bard, Hbent, Luke Warmwater101, Phatom87, Pinecar, RHaworth, Rror, Vishwas008, Vmahi9, 4 anonymous edits

Manual testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=437530057> *Contributors:* ArielGold, Ashish.agrawal17, DARTH SIDIOUS 2, Donperk, Eewild, Hairhorn, Iridescent, Kgarima, L Kensington, Meetusinh, Morrillonline, OIEnglish, Pinecar, Pinethicket, Predatoraction, Rwxrwxrwx, Saurabha5, Somdeb Chakraborty, SwisterTwister, Tumaka, 53 anonymous edits

Matrix of Pain *Source:* <http://en.wikipedia.org/w/index.php?oldid=275414965> *Contributors:* Andreas Kaufmann, Bill37212, Ctphonehome, Epastore, JonHarder, Kboswell, Lincher, Pinecar, SB Johnny, VRoscioli, 1 anonymous edits

Mauve (test suite) *Source:* <http://en.wikipedia.org/w/index.php?oldid=392403361> *Contributors:* Abdull, Atgreen, Fram, Govtpiggy, Gronky, Hervegirod, K14m, K14m-AWB, Ksnow, Pinecar, 11 anonymous edits

Metasploit Project *Source:* <http://en.wikipedia.org/w/index.php?oldid=446067462> *Contributors:* AKMask, AxelBoldt, Bjorkhaug, Byronknoll, Clément Brayer, Cybercobra, DanielPharos, Daverocks, Dbrn84, Den fjättrade ankan, Doug, Ergy, Erik J, Erik9, ErrantX, Faisal.akeel, Feezo, Frap, FrenchIsAwesome, Guy Harris, Hollowegian, IRWolfie-, Jmorgan, JonHarder, Jpbowen, Kingboyk, Kjak, K14m-AWB, Legotech, Loudsox, M. B., Jr., Manfromthemoon, Marc-André Aßbrock, Mcgyver5, Miserlou, Myc2001, Nneonneo, Operknockity, Pcap, Pouyana, Pradameinhoff, Quarl, ReCover, Rpyle731, Rtc, Rul3z, SF007, Sarkar112, Shadowjams, Sir Vicious, Storkk, TheBilly, TheIguana, Tothwolf, Travis.m.granvold, Varge0, VernoWhitney, Walter Görlitz, WibWobble, Zarkthehackeralliance, 111 anonymous edits

Microsoft Reaction Card Method (Desirability Testing) *Source:* <http://en.wikipedia.org/w/index.php?oldid=427716973> *Contributors:* AvicAWB, Malcolma, Mikec78uk, The Blade of the Northern Lights

Mobile Device Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=437659660> *Contributors:* ASmith1972, Belmont, Bovineone, Cander0000, Codetiger, Gregpaskal, JaGa, Pinecar, Schmloof, Swtechwr, Tsssyt, ZhonghuaDragon, 19 anonymous edits

Mockito *Source:* <http://en.wikipedia.org/w/index.php?oldid=430445645> *Contributors:* ChrisBowers, I42, Mawcs, MuffledThud, Sebastian.Dietrich, Tomrbj, Σ, 2 anonymous edits

Model-based testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=431719225> *Contributors:* Adivalea, Alvin Seville, Anthony.faucogney, Antti.huima, Bluemoose, Bobo192, Click23, Drilnoth, Ehheh, Eldad.palachi, FlashSheridan, Gaius Cornelius, Garganti, Hooperbloob, Jluedem, Jowler, Jzander, Kku, MDE, Mark Renier, MarkUtting, Mattisse, Mdd, Michael Hardy, Micskeiz, Mirko.conrad, Mjchonoles, MrOllie, Pinecar, Richard R White, S.K., Sdorance, Smartesting, Suka, Tatzelworm, Tedickey, Test-tools, That Guy, From That Show!, TheParanoidOne, Thv, Vonkje, Williamlasby, Yan Kuligin, Yxl01, 98 anonymous edits

Modified Condition/Decision Coverage *Source:* <http://en.wikipedia.org/w/index.php?oldid=445660744> *Contributors:* Andreas Kaufmann, Crazypete101, Freek Verkerk, Jabraham mw, Markiewp, Pindakaas, Vardhanw, 18 anonymous edits

Modularity-driven testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=318953592> *Contributors:* Avalon, Minnaert, Phanisrikar, Pinecar, 2 anonymous edits

Monkey test *Source:* <http://en.wikipedia.org/w/index.php?oldid=414521757> *Contributors:* Cimon Avaro, EagleFan, Ebelular, Lawilkin, Miq, Mirasmus, Nertz, Of, Pinecar, The Rambling Man, Thenickdude, Xyzzy n, 17 anonymous edits

Month of bugs *Source:* <http://en.wikipedia.org/w/index.php?oldid=392255440> *Contributors:* AlistairMcMillan, Bearcat, FMasic, Fiftyquid, FleetCommand, GNUtoo, Grafen, Manfromthemoon, Michaelsuarez, Ms2150, NapoliRoma, Ozzie The Owl, Pradameinhoff, RJFJR, Remember the dot, Robofish, ShelfSkewed, Yworo, Zvika, 8 anonymous edits

Mutation testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442722218> *Contributors:* Andreas Kaufmann, Antonielly, Ari.takanen, Brilesbp, Derek farn, Dogaroon, El Pantera, Felixwikhudson, Fuhghettaboutit, GiuseppeDiGuglielmo, Htmllapps, Jarfil, Jeffoffutt, JonHarder, Martpol, Mycroft.Holmes, Pieleric, Pinecar, Quuxplusone, Rohansahgal, Sae1962, Ustrnme h8er, Walter Görlitz, Wikid77, Yuejia, 47 anonymous edits

National Software Testing Laboratories *Source:* <http://en.wikipedia.org/w/index.php?oldid=367677041> *Contributors:* BuzzK, GiantSnowman, Greenshed, Nroets, Pinecar, Richhoncho, Spacepotato, 7 anonymous edits

NMock *Source:* <http://en.wikipedia.org/w/index.php?oldid=437791199> *Contributors:* Bearcat, D6, FlyingToaster, Kevin Rector, PamD, Santryl, 1 anonymous edits

Non-functional testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=408593175> *Contributors:* Addere, Burakseren, Dima1, JaGa, Kumar74, Mikethegreen, Ontist, Open2universe, P.srikanta, Pinecar, Walter Görlitz, 5 anonymous edits

Non-Regression testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=420213564> *Contributors:* Frap, GiacomoMS, SpiderJon

Operational Acceptance Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=444254073> *Contributors:* Amitg47, Badudoy, Bearcat, Bunnyhop11, Chris the speller, Katharineamy, Perdustin, Shire Reeve, Station1, Walter Görlitz, 5 anonymous edits

Oracle (software testing) *Source:* <http://en.wikipedia.org/w/index.php?oldid=414075847> *Contributors:* Andreas Kaufmann, Doug.hoffman, Hoo man, KathrynLybarger, Mchua, Propaniac, Updatehelper, Walter Görlitz, Xen0, Yfzhou, 11 anonymous edits

Original Software *Source:* <http://en.wikipedia.org/w/index.php?oldid=385014940> *Contributors:* Andy Dingley, Muhaned, Tumaka, 17 anonymous edits

Oulu University Secure Programming Group *Source:* <http://en.wikipedia.org/w/index.php?oldid=373851953> *Contributors:* Ari.takanen, LilHelpa, MLaub, PekkaPietikainen, Rich Farmbrough, ScottMHoward

Pair Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=425679879> *Contributors:* Andreas Kaufmann, Bjosman, LilHelpa, MrOllie, Neonleif, Prasantam, Tabletop, Universal Cereal Bus, Woohookitty, 9 anonymous edits

Parameter validation *Source:* <http://en.wikipedia.org/w/index.php?oldid=271014088> *Contributors:* Wikid77

Partial concurrent thinking aloud *Source:* <http://en.wikipedia.org/w/index.php?oldid=406062880> *Contributors:* Francesco.manno, GlassCobra, RHaworth, Simone.borsci, 4 anonymous edits

Penetration test *Source:* <http://en.wikipedia.org/w/index.php?oldid=443519540> *Contributors:* 2mcm, Aapo Laitinen, Adfectio, Ajcbylyth, Alejos, AlistairMcMillan, AllanBz, Amalas, Ankit bond2005, Atlant, Beland, Bobrayner, Bulzeeb, C. Foltz, Caesura, CanadianLinuxUser, Cantasta, Chris Chittleborough, Ckatz, Daemonaka, Davbradbury, Dcirovic, Derekslater, Doug Bell, Dubbon, Edward, Ehheh, Ermanon, Everyking, Fieldday-sunday, Frank Kai Fat Chow, Frap, Giraffedata, Godcast, GoneAwayNowAndRetired, Haidut, HamburgerRadio, Heqs, IReceivedDeathThreats, Ian13, Infosec23, Irishguy, Ivhtbr, JCarlos, Janet13, Jasburger, Jedi 001, Jinxpuppy, Johnuniq, JonHarder, Jpubal, KPH2293, Kaldosh, Killiondude, Kukini, Lsamaras11, M3tainfo, Manuelt15, Martinm79, Martinshp, Mild Bill Hiccup, Mile2, Mindmatrix, MI-crest, MrOllie, Nameless23, NeonMerlin, Newman12a, Nezzalisio, Njan, OmidPLuS, Ostendali, Page vandaliser, Patricioreyes, Pearle, Philip Trueman, Pieseckhi, Pinecar, Poeticos, PolarYukon, Pradameinhoff, Pxma, Qu3a, Radagast83, Ragib, Raistolo, Random name, Randy Johnston, Ransak, Riya.agarwal, Ronz, Rossbeth, Rzelnik, S charette, SF007, Securitytester, Sephiroth storm, Shamanchill, Shifat2sadi, Softtest123, SonOfBuzz, Special, SusanneOberhauser, Systemf, TechnoGuyRob, The Anome, The Thing That Should Not Be, Timc, ToddSweeney, Ultra Ine, Unicityd, VernoWhitney, Walter Görlitz, Whitehatnetizen, Wjejskenewr, Xample, Zabanio, 339 anonymous edits

Performance testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=433214533> *Contributors:* Al Lemos, Argyriou, Cfeet77, Closedmouth, Deicool, Dishit 17, Evenios, Hiqsynth, Hu12, Ijazmalik, Iulus Ascanius, Kuru, Mark.sbo, Mattisse, Michig, Mifter, Mywikicontribs, Physchim62, R'n'B, Rich Farmbrough, Rlsheehan, Sanjeev.behera, Sivagama sundari, Srikant.sharma, The Thing That Should Not Be, Webbyawards, Wilsonmar, Wrp103, Wtshymanski, YouWillBeAssimilated, 60 anonymous edits

PlanetLab *Source:* <http://en.wikipedia.org/w/index.php?oldid=410495300> *Contributors:* Alison, Anirvana, Bearcat, Beland, Big Brain, Calton, Daderot, Jashilo, Majorly, Melchoir, Mikeblas, Mindspillage, Mullern, Nethgirb, Nicholaskearns, Qutezuze, Rich Farmbrough, Sanmele, Santryl, Sbitc, SirFozzie, Sup7rstar, Ttimespan, Wongm, Xompanthy, Yserarau, Znwiki, ZombieDance, 12 anonymous edits

Playtest *Source:* <http://en.wikipedia.org/w/index.php?oldid=424825701> *Contributors:* Ajsh, Alansohn, Asla, Bmoross, Boarder8925, Classicrockfan42, Daveydwieb, Hooperbloob, Ksy92003, LOL, Muchness, Neoliminal, Percy Snoodle, Pinecar, Realkyhick, Woohookitty, 13 anonymous edits

Portability testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=415815752> *Contributors:* Andreas Kaufmann, Biscuitin, Cmdrjameson, Nibblus, OSborn, Pharos, Tapir Terrific, The Public Voice, 2 anonymous edits

Probe effect *Source:* <http://en.wikipedia.org/w/index.php?oldid=445158348> *Contributors:* A5b, Antonielly, Arch dude, Bwpach, Cjmclark, Danhash, DanielCD, Haze42, Kwertii, Tom Morris, 14 anonymous edits

Program mutation *Source:* <http://en.wikipedia.org/w/index.php?oldid=344608832> *Contributors:* Andreas Kaufmann, Antonielly, Ari.takanen, Brilesbp, Derek farn, Dogaroon, El Pantera, Felixwikihudson, Fuhghetaboutit, GiuseppeDiGuglielmo, Htmlapps, Jarfil, Jeffoffutt, JonHarder, Martpol, Mycroft.Holmes, Pieleric, Pinecar, Quuxplusone, Rohansahgal, Sae1962, Usrnme h8er, Walter Görlitz, Wikid77, Yuejia, 47 anonymous edits

Protocol implementation conformance statement *Source:* <http://en.wikipedia.org/w/index.php?oldid=409434321> *Contributors:* Alvin Seville, Behind The Wall Of Sleep, PamD, R'n'B, RHaworth

Pseudolocalization *Source:* <http://en.wikipedia.org/w/index.php?oldid=442625796> *Contributors:* A-)-Brunuś, Andy Dingley, Arithmandar, ArthurDenture, Autoterm, Bdjcomic, CyborgTosser, Dawn Bard, Gavrant, Günter Lissner, Josh Parris, Khazar, Kznf, Mboverload, Miker@sundialservices.com, Nilhenk, Pinecar, Pnm, Thumperward, Traveler78, Vipinhari, 10 anonymous edits

Pychecker *Source:* <http://en.wikipedia.org/w/index.php?oldid=435423369> *Contributors:* El Pantera

PyLint *Source:* <http://en.wikipedia.org/w/index.php?oldid=441267817> *Contributors:* El Pantera, InverseHypercube

User:Mbarberony/sandbox *Source:* <http://en.wikipedia.org/w/index.php?oldid=440684777> *Contributors:* Mbarberony

Software quality *Source:* <http://en.wikipedia.org/w/index.php?oldid=446124240> *Contributors:* 1ForTheMoney, Aislingdonnelly, Aitias, Alansohn, Alexf, AliveFreeHappy, Andreas Kaufmann, Ariconte, Ascánder, Bevo, Bigbluefish, Brent Gulanowski, Brian R Hunter, CRGreathouse, CanisRufus, Chatfacter, Chris the speller, Colonies Chris, Conan, Conortodd, Contributor124, Craigwb, Cybercobra, Dalvizu, Derek farn, Diomidis Spinellis, Discospinster, Dj stone, Dotxp, Doug.hoffman, Dwayne, Elendal, EoGuy, Erkan Yilmaz, Extrantit, GB fan, Gbolton, Gilloq, GregorB, Guybrush1979, Hooperbloob, Inray, J04n, JDBravo, JForget, JLaTondre, Jamelan, Jeff.fry, Jpbowen, Jsub, Jtigger, Kku, Krzysfr, Kuppuz, Kyokpae, Lalamax, Larsw, Latilgence, Lcarscad, M4gnum0n, MBisanz, MER-C, MONGO, Malleus Fatuorum, Manzee, Mark Renier, Matt Crypto, Mbarberony, Mdd, Mentifisto, Mhaitham.shammaa, Michedav, Mild Bill Hiccup, Mitch Ames, Mmcdougall, Mmerex, Mr.Muffet, MrOllie, Nesmojtar, Neverquick, No1lakersfan, NuclearWarfare, Oashi, On5deu, Paul.klinger, Pearle, Perfecto, Pgr94, Phase Theory, Piano non troppo, Project2501a, Promoa1, R'n'B, RJBurkhart3, Raoulduke47, RickClements, Rjwilmsi, Rp, Runderwo, S.K., Sangrolo, Sardanaphalus, Sean D Martin, Seanblanton, SkyWalker, Snigbrook, Snowolf, Someonesdad363616, Sp, SpuriousQ, Ste4k, Stephen, SunSwOrd, Suruena, Swtechwr, Tarret, Testerzy, Thowa, Thumperward, Thv, Tmaufer, ToSter, Tobias Bergemann, Viridae, Wizard191, Yangon, 205 anonymous edits

Recovery testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=410383530> *Contributors:* .digamma, DH85868993, Elipongo, Habam, LAAFan, Leandromartinez, Nikolay Shtabel, Pinecar, Rich257, Rjwilmsi, Vikramsharma13, 15 anonymous edits

Regression testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442842839> *Contributors:* 7, Abdull, Abhinavvaid, Ahsan.nabi.khan, Alan ffm, AliveFreeHappy, Amire80, Andrew Eisenberg, Anorpath, Antonielly, Baccyak4H, Benefactor123, Boongoman, Brenda Kenyon, Cabalamat, Cdunn2001, Chris Pickett, DRogers, Dacian.epure, Dee Jay Randall, Designatevoid, Doug.hoffman, Eewild, Elsendero, Emj, Enti342, Estyler, Forlornturtle, G0gegcsc300, Gregbard, Hadal, Henri662, HongPong, Hooperbloob, Iiiren, Jacob grace, Jwoodger, Kamarou, Kesla, Kmincey, L Kensington, Labalius, LandruBek, Luckydrink1, MER-C, Marijn, Mariotto2009, Matthew Stannard, Maxwellb, Menzogna, Michaelas10, Michig, MickeyWiki, Mike Rosoft, MikeLynch, Msillil, NameIsRon, Neile, Neurolysis, Philipchiappini, Pinecar, Qatutor, Qfissler, Ravialluru, Robert Merkel, Rsavenkov, Ryans.ryu, S3000, Scoops, Snarius, Spock of Vulcan, SqueakBox, Srittau, Strait, Svick, Swtechwr, Throwaway85, Thv, Tobias Bergemann, Tobias Hoevekamp, Toon05, Walter Görlitz, Will Beback Auto, Wlievens, Zhenqinli, Zvn, 171 anonymous edits

Release engineering *Source:* <http://en.wikipedia.org/w/index.php?oldid=430380333> *Contributors:* 041744, Abhinavvaid, Andreas Kaufmann, Anvish, Dagg, Daniel.Cardenas, Freshbaked, Furrykef, Gwhodgson, Hashar, Imeshev, JCreashaw, JonHarder, Jwarhol, Lightblade, LilHelpa, Longhair, Lycurgus, Mdd, Mikeblas, Milkfish, N-miyu, NuclearWinner, Sardanaphalus, Sivala, Stevage, Tedickey, Tracyragan, Wolfdancer, 25 anonymous edits

Retrofits in testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=419952257> *Contributors:* Aarunkadam, Katharineamy, Malcolm, Racklever

Reverse semantic traceability *Source:* <http://en.wikipedia.org/w/index.php?oldid=412620894> *Contributors:* Andreas Kaufmann, Foobarnix, Gaius Cornelius, Jpbowen, Olga.Petrus, Pinecar, RHaworth, 3 anonymous edits

Risk-based testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=442829802> *Contributors:* Andreas Kaufmann, DRogers, Deb, Gilliam, Henri662, IQDave, Lorezsky, Paulgerrard, Ronhjones, Tdjones74021, VestaLabs, 14 anonymous edits

Robustness testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=420294234> *Contributors:* Ari.takanen, Cesium 133, Elektrik Shoos, Malcolm, Micskeiz, Tobias Bergemann, 2 anonymous edits

San Francisco depot *Source:* <http://en.wikipedia.org/w/index.php?oldid=362281144> *Contributors:* Andreas Kaufmann, Auntof6, Centrx, DRogers, EagleFan, Fabrictramp, PigFlu Oink, Pinecar

Sandbox (software development) *Source:* <http://en.wikipedia.org/w/index.php?oldid=443312888> *Contributors:* 22breeze, Ais523, Akkida, Andrewmc123, Angosso, Antonio Giusti, Arthna, Benlisquare, Bxstudent, Cbratschi, Crazyaman, DARTH SIDIOUS 2, Da monster under your bed, Desaigamon, Download, Drilnoth, Eklawyers, Fraggle81, Frosty0814snowman, Gsf, IRP, Intelligentium, Iridescent, J.delaney, JCO312, JForget, JodyB, JoeOpacity, Jusdafax, Kerii57, Lucas Blade, LuckyWolf19, Luke1138, MER-C, Majorly, Mattakagod, Nantipov, Nascar1996, NateEag, NerdyScienceDude, Nurg, Persian Poet Gal, Prakashxclusive, PrimeHunter, Qwertymatt15, Raulgvazquez, Rizome2, Rocknmandan, Rror, RyanCross, SMCandlish, SVG, SchuminWeb, Shappy, Snbaka, Soandos, Str4nd, Tempodivalse, The Thing That Should Not Be, TheParanoidOne, Tide rolls, Tobias Bergemann, TonyW, Trotter, UncleDouggie, Van der Hoorn, Vegpuff, Versus22, Winchelsea, ZX81, 104 anonymous edits

Sanity testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=438755122> *Contributors:* Andycjp, Arjayay, Chillum, Chris Pickett, Closedmouth, D4g0thur, Dysprosia, Fittysix, Fullstop, Gorank4, Haus, Histriorn, Itai, JForget, Kaimiddleton, Karada, Kingpin13, LeaW, Lechatjaune, Lee Daniel Crocker, Martinwguy, Matma Rex, Melchoir, Mikewalk, Mild Bill Hiccup, NeilFraser, Nunh-huh, Oboler, PierreAbbat, Pinecar, Pinethickett, Polluks, R'n'B, Ricardol, Rrburke, Saberywn, Sietsje Snel, SimonTrew, Strait, Stratadrake, UlrichAAB, Verloren, Viriditas, Walter Görlitz, Webinonline, Wikid77, 98 anonymous edits

Scalability testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=411550068> *Contributors:* GregorB, JaGa, Kumar74, Malcolma, Methylgrace, Pinecar, 6 anonymous edits

Scenario testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=415899231> *Contributors:* Abdull, Alai, Bobo192, Brandon, Chris Pickett, Epim, Hu12, Karbinski, Kingpin13, Kuru, Pinecar, Ronz, Shepard, Walter Görlitz, 20 anonymous edits

Security bug *Source:* <http://en.wikipedia.org/w/index.php?oldid=421871632> *Contributors:* Josephgrossberg, Jruderman, LarryHughes, Mysdaao, Pastore Italy, Pinecar, Robofish, Schmoof, 7 anonymous edits

Security testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=445168305> *Contributors:* Aaravind, Andreas Kaufmann, Bigtimepeace, Bwpach, ConCompS, DanielPharos, David Stuble, Dxdwell, Epr123, Gardener60, Gavenko a, Glane23, ImperatorExercitus, JonHarder, Joneskou, Kinu, Lotje, MichaelBillington, Pinecar, Ravi.alluru@applabs.com, Shadowjams, SoftwareTest1, Someguy1221, Spitfire, Stenaught, ThisIsAce, Uncle Milty, WereSpielChequers, 98 anonymous edits

Semantic decision table *Source:* <http://en.wikipedia.org/w/index.php?oldid=429010101> *Contributors:* Airsplit, CommonsDelinker, Drbreznjev, Malcolma, Woohookitty, Yan.tang

Serenity Code Coverage *Source:* <http://en.wikipedia.org/w/index.php?oldid=420041622> *Contributors:* Michael.couck, MuffledThud, Ptrb, Wilhelmina Will

Session-based testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=446122617> *Contributors:* Alai, Bjosman, Chris Pickett, DRogers, DavidMJam, Jeff.fry, JenKilmer, JulesH, Pinecar, Walter Görlitz, WikHead, 14 anonymous edits

SignationTF *Source:* <http://en.wikipedia.org/w/index.php?oldid=374639015> *Contributors:* Alvin Seville, Grace yuan, Graeme Bartlett, Malcolma, Your Lord and Master

Smoke testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=446151551> *Contributors:* Adam Zábranský, Akadruid, Alan Liefing, AmiDaniel, Amty4all, Atlant, Balderdash707, Beland, BenBerry, BenFrantzDale, Beryn68, Busy Stubber, Charles Matthews, Chiverisimilidude, Chris Pickett, Cmdrjameson, Culix, DMG413, Dagostinelli, Damian Yerrick, Dee Jay Randall, Difu Wu, Drpickem, Erkan Yilmaz, FreplySpang, G-J, Gbleem, Gene Nygaard, Gordon Ecker, Harda, Harrywferguson, Hooperbloob, Jaksmata, Jeffshantz, Jnegomir, Jojhutton, JukoFF, JulesN, Lee Daniel Crocker, Liftarn, LittleDan, MBisanz, Mbrooks, Michal Nebyla, Miker@sundialservices.com, MrInitialMan, OIEnglish, Pinecar, Quiesagua, Ravialluru, Rkumar99, Romanski, Rpresser, SaintCahier, Sam Hovevar, Sbrockway, Skarkkai, Stratadrake, Super j dynamite, Tetsuo, Texture, ThomasOwens, Timothee, Tohd8BohathuGh1, Tom Lougheed, Trensher, TutterMouse, Ummit, Walter Görlitz, Windolene, Zummed, 135 anonymous edits

Soak testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=445483615> *Contributors:* A1r, DanielPharos, JPFitzmaurice, JnRouvignac, Mdd4696, Midlandstoday, P mohanavan, Pinecar, 11 anonymous edits

Soapsonar *Source:* <http://en.wikipedia.org/w/index.php?oldid=445258863> *Contributors:* Bostonmace, Chzz, Elwikipedista, Jonkerz, Malcolmx15, Mlaffs, YUL89YYZ, Yamachiro, 3 anonymous edits

SOASTA *Source:* <http://en.wikipedia.org/w/index.php?oldid=408376680> *Contributors:* AnnaFinotera, Bearcat, Beefyt, Jahub, John Vandenberg, LinguistAtLarge, Malcolma, Nyttend, Pinecar, Rich Farmbrough, Rjwilmsi, Ron Ritzman, SamJohnston, 1 anonymous edits

Software performance testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=444211955> *Contributors:* AMbroodEY, AbsolutDan, Alex Vinokur, Andreas Kaufmann, Apodelko, Argyniou, Armadillo-eleven, Bourgeoispy, Brian.a.wilson, Burakseren, Cit helper, Ckoenigsberg, Coroberti, D6, David Johnson, Davidschmelzer, Deicool, Dhiraj1984, Dwisser, Edepriest, Freek Verkerk, Ghewgill, Gnowor, Grotendeels Onschadelijk, Gururajs, Hooperbloob, Hu12, Ianmolynz, Iulus Ascanius, JaGa, Jdlow1, Jeremy Visser, Jewbacca, Jncraton, Katremer, Kbustin00, Ken g6, KnowledgeOfSelf, M4gnum0n, MER-C, Maimai009, Matt Crypto, Matthew Stannard, Michig, MrOllie, Mrmatiko, Msadler, Muhanes, Mywikicontribs, Nono64, Notinasnoid, Oliver Lineham, Optakeover, Pinecar, Pratheepraj, Ravialluru, Raysecurity, Rjwilmsi, Robert Merkel, Ronz, Rsbarber, Rstens, Sebastian.Dietrich, ShelfSkewed, Shimser, Shirtwaist, Shoefdeath, SimonP, Softlogica, SunSw0rd, Swtechwr, Timgurto, Veinor, Versageek, Wahab80, Walter Görlitz, Weregerbil, Wilsonmar, Wizzard, Wktsugue, Woohookitty, Wselph, 235 anonymous edits

Software testability *Source:* <http://en.wikipedia.org/w/index.php?oldid=411787003> *Contributors:* Contributor124, Favonian, Oscartheac, UlrichAAB, Walter Görlitz, 4 anonymous edits

Software testing controversies *Source:* <http://en.wikipedia.org/w/index.php?oldid=431773787> *Contributors:* Andreas Kaufmann, Derelictfrog, JASpencer, PigFlu Oink, Pinecar, Softtest123, 4 anonymous edits

Software testing life cycle *Source:* <http://en.wikipedia.org/w/index.php?oldid=441489181> *Contributors:* Dianna, Fabrictramp, Katharineamy, Malcolma, MaterialsScientist, Mdwh, Socialservice, Stifle, Vishwas008, 18 anonymous edits

Software testing outsourcing *Source:* <http://en.wikipedia.org/w/index.php?oldid=408750432> *Contributors:* Algebraist, Dawn Bard, Discospinster, Elagatis, Hu12, Kirk Hilliard, Lolawrites, NewbieIT, Piano non troppo, Pinecar, Pratheepraj, Promoa1, Robofish, TastyPoutine, Tedickey, Tesstty, Woohookitty, 15 anonymous edits

Software Testing, Verification & Reliability *Source:* <http://en.wikipedia.org/w/index.php?oldid=409149921> *Contributors:* Crusio, Jpbowen, Kajervi, MarcoLittel

Software verification *Source:* <http://en.wikipedia.org/w/index.php?oldid=435881015> *Contributors:* Academy633, AjAlouds, A117, Andreas Kaufmann, Boddahboy, CoderGnome, Cryptic, Dalesgay, DéRahier, Enoch the red, Fdotalavi, Hooperbloob, Indon, Jeff3000, Jevon, MER-C, Manop, Pinecar, Schmittey, Sindheeraj, Sozin, Thv, Zuidervled, 24 anonymous edits

Sputnik (JavaScript conformance test) *Source:* <http://en.wikipedia.org/w/index.php?oldid=445849333> *Contributors:* Twolfblake, Binaryguru, Daniel Hen, Desplow, DonDublon, Mabdul, Nyttend, Pointillist, Waldir, 96 anonymous edits

STAR (Conference) *Source:* <http://en.wikipedia.org/w/index.php?oldid=349579144> *Contributors:* Conferensum, Martijn Hoekstra, Mpilaeten

Stream X-Machine *Source:* <http://en.wikipedia.org/w/index.php?oldid=416402765> *Contributors:* AJHSimons, Mike.stannett, Rich Farmbrough, 1 anonymous edits

Stress testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=438240020> *Contributors:* .K, Alenander, Alex.g, Alpinwolf, Attilios, Ayucat, Bhadani, Btyner, Cfeet77, Colchicum, Con-struct, CyborgTosser, Czambra, Discospinster, Donald Albury, Eiland, Enzo Aquarius, Freek Verkerk, G3pro, Gotest01, Hooperbloob, Hu12, Ileshko, JMSwtik, Jahasloth, JamesBrownJr, Jareha, Jedidan747, Jeffrey Mall, Jemtreadwell, Jim.henderson, Johnuniq, Keilana, Kkorpinen, Ktsummer, M4gnum0n, MALvis, Madannus, Michig, Nbarth, Nurg, OIEnglish, Pats1, Pinecar, Praveen98480, RP459, Rlsheehan, Sebesta, Shijaz, SimonP, ThirteenthGreg, Tripledot, Tuvsy, Una Smith, Walkingstick3, Walter Görlitz, WildWildBil, Zaphraud, Zhenqinli, 71 anonymous edits

Stress testing (software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=434725382> *Contributors:* Brian R Hunter, Con-struct, CyborgTosser, Hu12, Pinecar, Tobias Bergemann, Trevj, 11 anonymous edits

System integration testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=444023688> *Contributors:* Aliasgarshakir, Andreas Kaufmann, Andrewmillen, Anna Lincoln, AvicAWB, Barbzie, Bearcat, Charithk, DRogers, Fat pig73, Flup, Gaius Cornelius, JeromeJerome, Jpbowen, Kku, Mawcs, Mikethegreen, Panchitaville, Pinecar, Radagast83, Rich Farmbrough, Rwww, Walter Görlitz, 31 anonymous edits

System testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=433091657> *Contributors:* A bit iffy, Abdull, AliveFreeHappy, Anant vyas2002, AndreChou, Argon233, Ash, Bex84, Bftsg, BiT, Bobo192, Ccompton, ChristianEdwardGruber, Closedmouth, DRogers, Downsize43, Freek Verkerk, GeorgeStepanek, Gilliam, Harveysburger, Hooperbloob, Ian Dalziel, Jewbacca, Kingpin13, Kubigula, Lauwerens, Manway, Michig, Mpilaeten, Myhister, NickBush24, Philip Trueman, Pinecar, RCHenningsgard, RainbowOfLight, Ravialluru, Ronz, Solde, Ssweeting, Sufusion of Yellow, SusanLarson, Thv, Tmopkisin, Vishwas008, Vmah9, Walter Görlitz, Whckwo, Zhenqinli, 128 anonymous edits

Tessy (Software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=407487775> *Contributors:* Alvin Seville, Davidgilestex, Frap, GabiS, Graeme Bartlett, MuffledThud, 1 anonymous edits

Test Anything Protocol *Source:* <http://en.wikipedia.org/w/index.php?oldid=445662213> *Contributors:* Andreas Kaufmann, AndyArmstrong, BrotherE, Brunodepaulak, Frap, Gaurav, Justatheory, Mindmatrix, Pinecar, RJHerrick, Schwern, Shlomif, Tarchannen, Thr4wn, Wrelwser43, Ysth, 30 anonymous edits

Test automation *Source:* <http://en.wikipedia.org/w/index.php?oldid=445305303> *Contributors:* Snizza, 83nj1, ADobey, Abdull, Akr7577, Alaattinoz, AliveFreeHappy, Ameya barve, Ancheta Wis, Andy Dingley, Ankurj, Anupam naik, Apparition11, Asashour, Ash, Auntof6, Bbryson, Benjamin Geiger, Bhagat.Abhijeet, Bigtwilkins, Caltas, Carioca, Checkshirt, Chrisbepost, Christina thi, CodeWonk, DARTH SIDIOUS 2, DRogers, Dbelhumeur02, DivineAlpha, Dreflymac, Eaozens, EdwardMiller, Egivoni, ElfriedeDustin, Elipongo, Enoch the red, Excirial, Faris747, Ferpectionist, FlashSheridan, Flopsy Mopsy and Cottonmouth, Florian Huber, Fumitol, G0gogesc300, Gaggargwal2000, Gherget, Gibs2001, Gmacgregor, Goutham, Grafen, Harobed, Hatch68, Helix84, Hesa, Heydaysoft, Hooperbloop, Hswiki, Hu12, Ixfid64, JASpencer, JamesBWatson, Jkoprax, Johnuniq, Jpg, Kumarsameer, Kuru, Ldimaggi, M4gnum0n, MC10, MER-C, Marasmusine, Mark Kilby, Marudubshinki, Matthewedwards, Mdanel, MendipBlue, Michael Bernstein, Morrillonline, MrOllie, Nima.shahhosini, Nimowy, Notinasnaid, Octoferret, Ohnoitsjamie, OracleDBGuru, PeterBizz, Pfhjv0, ProfessionalTST, Qatutor, Qlabs impetus, Qtpautomation, Qwyrxian, R'n'B, RHaworth, Radagast83, Radiant!, Radiostationary, Raghublr, Raymondlafourchette, Rich Farmbrough, RichardHoultz, Rickjpellg, Rjwilmsi, Robertvan1, Robinson Weijman, Ryadav, Ryepie, SSmithNY, Sbono, ScottSteiner, Seaphoto, Shankar.sathiamurthi, Shijuraj, Shlomif, SoCalSuperEagle, Softwaretest1, Srideep TestPlant, Ssingaraju, SteveLoughran, Sundaramkumar, Swtechrw, Testautomator, Thv, Ttrevers, Tumaka, Tushar291081, Vadimka, Veledan, Versageek, Vogel, Waikh, Walter Görlitz, Webbbbbbber, Winmacro, Wrp103, Yan Kuligin, ZachGT, Zorgon7, सनेन कुमार ढकाल, 305 anonymous edits

Test automation framework *Source:* <http://en.wikipedia.org/w/index.php?oldid=426570771> *Contributors:* Abdull, Aby74, Akr7577, A195521, AliveFreeHappy, Andy Dingley, Anshoorora, Apparition11, Chrisbepost, Closedmouth, Drpaule, Excirial, Flopsy Mopsy and Cottonmouth, Gibs2001, Heydaysoft, Homfri, Iridescent, JamesBWatson, Jonathan Webley, LegendGamer, Mitch Ames, Mountk2, Nalinnew, Oziransky, Paul dexus, Peneh, PeterBizz, Pinecar, Qlabs impetus, RHaworth, Regancy42, Rsciaccio, Sachxn, Sbasan, SerejkaVS, SteveLoughran, Vishwas008, Walter Görlitz, West.andrew.g, 55 anonymous edits

Test automation management tools *Source:* <http://en.wikipedia.org/w/index.php?oldid=417766520> *Contributors:* PeterBizz, RadioFan, SunCountryGuy01, Walter Görlitz

Test bench *Source:* <http://en.wikipedia.org/w/index.php?oldid=437094164> *Contributors:* Abdull, Ali65, Amitusain, Arch dude, Dolovis, E2eamon, FreplySpang, J. Sparrow, Joe Decker, Pinecar, Remotelysensed, Rich Farmbrough, Singamayya, Testbench, Tgruwel, 12 anonymous edits

Test case *Source:* <http://en.wikipedia.org/w/index.php?oldid=443526257> *Contributors:* AliveFreeHappy, Allstarecho, Chris Pickett, ColBatGuano, Cst17, DarkBlueSeid, DarkFalls, Darth Panda, Eastlaw, Flavioxavier, Freek Verkerk, Furrykef, Gothmog.es, Hooperbloop, Igyy402, Iondiode, Jtowler, Jwh335, Jwoodger, Kevinmon, LeaveSleaves, Lenoxus, Magioladitis, Maniacs29, MaxHund, Mdd, Merutak, Mr Adequate, MrOllie, Nibblus, Niri.M, Nmthompson, Pavel Zubkov, Peter7723, Pilaf, Pinecar, PrimeObjects, RJFJR, RainbowOfLight, RayAYang, Renu gautam, Sardanaphalus, Sciriurinae, Sean D Martin, Shervinafshar, Srikaana123, Suruena, System21, Thejesh.cg, Thorncrag, Thv, Tomaxer, Travelbird, Vellela, Vikasbucha, Walter Görlitz, Wernight, Yenth, Zack wadghiri, 175 anonymous edits

Test data *Source:* <http://en.wikipedia.org/w/index.php?oldid=444652160> *Contributors:* AlexandrDmitri, Alvestrand, Craigwb, Fg2, JASpencer, Nnesbit, Onorem, Pinecar, Stephenb, Uncle G, 10 anonymous edits

Test design *Source:* <http://en.wikipedia.org/w/index.php?oldid=401407763> *Contributors:* Coolug, JnRouvignac, Walter Görlitz, WereSpielChequers, 1 anonymous edits

Test Double *Source:* <http://en.wikipedia.org/w/index.php?oldid=432912468> *Contributors:* Graeme Bartlett, Mawcs, Melaen, MuffledThud, Random User 937494, Tomrbj, 1 anonymous edits

Test effort *Source:* <http://en.wikipedia.org/w/index.php?oldid=422710539> *Contributors:* Chemuturi, Chris the speller, Contributor124, DCDuring, Downsize43, Erkan Yilmaz, Furrykef, Helodia, Lakeworks, Mr pand, Notinasnaid, Pinecar, Ronz, 10 anonymous edits

Test execution engine *Source:* <http://en.wikipedia.org/w/index.php?oldid=393388726> *Contributors:* Abdull, Ali65, Andreas Kaufmann, Cander0000, ChildofMidnight, Fabrictramp, Grafen, Rontaih, Walter Görlitz, 1 anonymous edits

Test harness *Source:* <http://en.wikipedia.org/w/index.php?oldid=430657567> *Contributors:* Abdull, Ali65, Allen Moore, Avalon, Brainwavz, Caesura, Caknuck, Calréfa Wéná, DenisYurkin, Downtown dan seattle, Dugrocker, Furrykef, Greenrd, Kgaughan, Pinecar, SQAT, Tony Sidaway, Wlievens, 43 anonymous edits

Test management *Source:* <http://en.wikipedia.org/w/index.php?oldid=441349197> *Contributors:* Andreas Kaufmann, B, Bhayduk, D.brodale, Egevaldo, Enochlau, Frap, Gardar Rurak, Halloleo, Havlatm, Henri662, Jinglec7, Jkoprax, Moe Epsilon, Mortense, Piano non troppo, Pinecar, Pinkadelica, Rlsheehan, Softwareqaguru1, Suhashpatil, Walter Görlitz, 49 anonymous edits

Test Management Approach *Source:* <http://en.wikipedia.org/w/index.php?oldid=432813761> *Contributors:* Donzall, Dstowdam, Euchiasmus, Marcelo Pinto, Michael.oconnell1, Mild Bill Hiccup, Olekva, 11 anonymous edits

Test plan *Source:* <http://en.wikipedia.org/w/index.php?oldid=440831109> *Contributors:* -Ril-, Aaronbrick, Aecis, Alyna Kasmira, AndrewStellman, AuburnPilot, Bashnya25, Charles Matthews, Craigwb, Dave6, Downsize43, Drable, E Wing, Foobaz, Freek Verkerk, Grantmidnight, Hennessey, Patrick, Hongooi, Icbkr, Ismarc, Jaganathcfs, Jason Quinn, Jeff3000, Jgorse, Jla04, Ken tabor, Kindx, Kitdaddio, LogoX, M4gnum0n, MarkSweep, Matthew Stannard, Mellissa.mcconnell, Michig, Mk*, Moonbeachx, NHSavage, NSR, Niceguyedc, OllieFury, Omicronpersei8, OndraK, Oriwall, Padma vgp, Pedro, Pinecar, RJFJR, RL0919, Rlsheehan, Roshanoimam, Rror, SWAdair, Schmiteye, Scope creep, Shadowjams, SimonP, Stephenb, Tgeairn, The Thing That Should Not Be, Thunderwing, Thv, Uncle Dick, Wacko, Wagers, Walter Görlitz, Yparedes, 321 anonymous edits

Test script *Source:* <http://en.wikipedia.org/w/index.php?oldid=396053008> *Contributors:* Alai, Eewild, Falterion, Freek Verkerk, Hooperbloop, JLaTondre, JnRouvignac, Jruska, Jwoodger, Michig, PaulMEdwards, Pfhjv0, Pinecar, RJFJR, Rechandra, Redrocket, Sean.co.za, Sujaiakareik, Teiresias, Thv, Ub, Walter Görlitz, 28 anonymous edits

Test strategy *Source:* <http://en.wikipedia.org/w/index.php?oldid=441870269> *Contributors:* AlexWolfx, Autoerrant, Avalon, BartJandeLeuw, Christopher Lamothe, D6, Downsize43, Fabrictramp, Freek Verkerk, HarlandQPitt, Henri662, Jayaramg, John of Reading, Liheng300, LogoX, M4gnum0n, Malcolma, Mandarhambir, Mboverload, Michael Devore, Pinecar, RHaworth, Rpyle731, Shepard, Walter Görlitz, 69 anonymous edits

Test stubs *Source:* <http://en.wikipedia.org/w/index.php?oldid=357487595> *Contributors:* Andreas Kaufmann, Chiefhuggybear, Christianvinter, Meredith K, Thisarticleisastub, Tomrbj, 1 anonymous edits

Test suite *Source:* <http://en.wikipedia.org/w/index.php?oldid=440123583> *Contributors:* A-hiro, Abdull, Alai, Andreas Kaufmann, CapitalR, Denispir, Derek farn, FreplySpang, JzG, KGasso, Kenneth Burgener, Lakeworks, Liao, Martpol, Newman.x, Pinecar, Tomjenkins52, Unixtastic, VasilievVV, Walter Görlitz, 25 anonymous edits

Test Template Framework *Source:* <http://en.wikipedia.org/w/index.php?oldid=410042882> *Contributors:* Auntof6, Mcristia, Pnm, 17 anonymous edits

Test Vector Generator *Source:* <http://en.wikipedia.org/w/index.php?oldid=174356865> *Contributors:* Alvestrand, Lordkilgour, TheParanoidOne

Test-driven development *Source:* <http://en.wikipedia.org/w/index.php?oldid=445516746> *Contributors:* Israghavan, Achorny, AliveFreeHappy, Aiksentrs, Anorthup, AnthonySteele, Antonielly, Asgeirn, Astaines, Attilios, Autarch, AutumnSnow, Bewhite, Beland, CFMWiki1, Calréfa Wéná, Canterbury Tail, Chris Pickett, Closedmouth, Craig Stuntz, CraigTreptow, DHGarrette, Dally Horton, David-Sarah Hopwood, Deuxpi, Dhdblues, Dougluce, Download, Downsize43, Dtimilano, Dlugosz, Ed Poor, Edaelon, Ehheh, Emurphy42, Enochlau, Eurlief, Excirial, Faught, Fbeppler, Fre0n, Furrykef, Gakrivav, Gary King, Geometry.steve, Gigi fire, Gishu Pillai, Gmcrews, Gogo Dodo, Hadal, Hagai Cibulski, Hariharan wiki, Heirpixel, Hzhbcl, JDBravo, JLaTondre, JacobProffitt, Jglynn43, Jleedev, Jonb ee, Jonkpa, Jpalm 98, Jrzv, Kbdank71, Kellen*, KellyCoinGuy, Kevin Rector, Khalid hassani, Kristjan Wager, Krzyk2, Kvdveer, LeaveSleaves, Lenin1991, Lumberjake, Madduck, Mark Renier, Martial75, Martinig, MaxSem, Mberteig, Mboverload, Mckoss, Mdd, MeUser42, Mhhanley, Michael miceli, Michig, Middyayexpress, Mkarlesky, Mksingha, Mnorbury, Mortense, Mosquitopus, Mossd, Mr2001, MrOllie, Nigelj, Nohat, Notnoisy, Nuggetboy, Ojcit, Oligomous, OnSdeu, Pocklandsapanaway, Patrickdepinguin, Pengo, PhilipR, Philip2005, Pinecar, PradeepAryal109, R. S. Shaw, Radak, Raghunathan.george, RickBeton, RoyOsheroove, Rulesdoc, SAE1962, Sam Hovevar, Samwashburn3, San chako, Sanchom, SethTissue, Shadowjams, SharShar, Shenme, Shyam 48, SimonP, St.General, Stemed, SteveLoughran, Sullivan.t, Supreme Deliciousness, Sverdrup, Svick, Swadsen, Szwejkce, TakuyaMurata, Tedickey, Themacoby, Thumperward, Tobias Bergemann, Topping, Trum123, Underpants, V6Zi34, Virgiltrasca, WLU, Walter Görlitz, Waratah, Wikid77, Xagronaut, Олександр Кравчук, 408 anonymous edits

Test-Driven Development by Example *Source:* <http://en.wikipedia.org/w/index.php?oldid=376385130> *Contributors:* Andreas Kaufmann, Autarch, D6, Ed Poor, El Pantera, GoingBatty, Haemo, Pinecar, RenukaM, Tierlieb, 1 anonymous edits

Testbed *Source:* <http://en.wikipedia.org/w/index.php?oldid=421858587> *Contributors:* Aintneo, Amtiss, Bwpach, Chuunen Baka, Dilumb, Dmatahari, Docboat, Favonian, Julian Mendez, Kgf0, Mabdul, Mcfly85, Ninly, P.Ramakrishna Raju, Pinecar, Pinethicket, Rentaferrret, Salt3d, Stustu12, Sycthos, Underpants, Walter Görlitz, Whitepaw, Winterstestbed, 31 anonymous edits

Tester driven development *Source:* <http://en.wikipedia.org/w/index.php?oldid=340119755> *Contributors:* Arneeri, BirgitteSB, Chris Pickett, Fram, Gdavidp, Int19h, Josh Parris, Mdd, Pinecar, Sardanaphalus, Smjg, 11 anonymous edits

Tester forum *Source:* <http://en.wikipedia.org/w/index.php?oldid=306970221> *Contributors:* Alvestrand, CiTrusD, Malcolma, Paul6feet1, 2 anonymous edits

Testing as a service *Source:* <http://en.wikipedia.org/w/index.php?oldid=434023578> *Contributors:* Dannycrone, Johnniq, Malcolm, MrOllie, RHaworth, Rettetast, Rich Farmbrough, Ryan lee james, Tashif, 21 anonymous edits

Testing Maturity Model *Source:* <http://en.wikipedia.org/w/index.php?oldid=437635949> *Contributors:* Andesh87, Auntof6, Crest.boy, John a s, Kuyabribri, LilHelpa, Matrask, Matthew Stannard, Rich Farmbrough, The Man in Question, TimMoore, 4 anonymous edits

Testware *Source:* <http://en.wikipedia.org/w/index.php?oldid=372600851> *Contributors:* Andreas Kaufmann, Assadmalik, Avalon, Gzkn, Robofish, SteveLoughran, Wireless friend, ZhonghuaDragon, 7 anonymous edits

Think aloud protocol *Source:* <http://en.wikipedia.org/w/index.php?oldid=423009315> *Contributors:* Akamad, Angela, Aranel, Calebjc, Crònica, DXBari, Delldot, Diego Moya, Dragice, Hetar, Icairns, Jammycaketin, Khalid hassani, Manika, Ms2ger, Nuggetboy, Ofol, Ohnoitsjamie, PeregrineAY, Pinecar, Robin S, Robksw, Ronz, Shanes, Shevek57, Simone.borsci, Suruena, TIY, Technopat, Tillwe, Wik, Zojiji, Zunk, 23 anonymous edits

Tiger team *Source:* <http://en.wikipedia.org/w/index.php?oldid=426280009> *Contributors:* 2fs, Andreas Kaufmann, Andux, Azumanga1, Beamjockey, Beepbopbeepbop, Below Iover-1, Bookandcoffee, Brockert, Caesura, DMCer, Da monster under your bed, Dansdoinit, EggRoll, Emurphy42, Flopsy Mopsy and Cottonmouth, Good Olfactory, Gramarye1971, Halda, Harryzilber, Hirsutism, Ida Shaw, Idont Havename, Infeh, JDspeeder1, Jinxpuppy, Jnc, Joema, Julesd, LightSpirit, Liko81, M3tainfo, M3tainfo, Magnius, Meelar, MementoVivere, Michael Hardy, Mullerpaulum, NeonMerlin, Noclevername, Pengo, Pinecar, Rifleman 82, Rogerborg, Rtc, Sandeepmcpd, Sephiroth storm, Shimaspawn, Sin-man, Sullyss73, Template namespace initialisation script, VockingRandal, Wikipetproject, 50 anonymous edits

Tosca (Software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=444073353> *Contributors:* Bunnyhop11, DanielPharos, Decamerone, Jkoprax, Muhandes, Steinstbauer, Welsh, Woohookitty, 2 anonymous edits

TPS report *Source:* <http://en.wikipedia.org/w/index.php?oldid=441081923> *Contributors:* 1980fast, 21655, A Clown in the Dark, Abelson, Adagio Cantabile, Admrboltz, Afiler, Aksnit, Anna megan123, Antonbrandt, Arklenao2, Arxiloxos, Ask123, Atehoffman, Avanu, Awm, B7T, BabuBhatt, Basilargento, Bobo192, Branddobbe, Canislupusarctos, ChinaMike, Cholling, Clement Cherlin, Cnota, Crobichaud, Cyberia23, Cyclonius, Daveblack, Davehi1, Dismas, Djswiss1, Dondegroovily, Dreaded Walrus, Ed Fitzgerald, Ekbeale, Ensniffraff, Evil Monkey, Ewlyhaoooom, Fullmetal2887, Gavreh, Gezzas Man, Gnsmsk, GreenReaper, Haakon, Hahhchen, Hm2k, Hooperbloob, Horkana, IanManka, IwRnHaA, JDspeeder1, Jac16888, James.Denholm, Jdottraub, JensRex, Jivlain, Jmchuff, JohnnyPants, Josiah Rowe, Jotdeh, JustPhil, Kalmbach, Karey, Katieh5584, Kazrak, Keithmahoney, Kevinconroy, Komjdp, Landoltjp, Lheaom, Lockesdonkey, Lpasquer, Luke4545, Maduskis, Martarius, Rev, MementoVivere, Merranvo, Mike Payne, Minesweeper, Mnedmonson, Nilicule, Not Diablo, Omhafaieio, One, PaulHanson, Phoenixrod, Pmchesun, RedSpruce, Reluctantpopstar, RevRagnarok, Rgoodermote, Robofish, Rsnell72, Sarcacinjinja, SchuminWeb, Statsone, Superchink, Swayid, TabsAZ, Tamajared, Tealwip, Th1rt3en, Thumperward, Timvasquez, ToastyKen, Twaz, UberMan5000, VictorianMutant, Vsync, Waldir, Wyrdbyr, X1011, Xaosflux, Zoef1234, Zoganes, 219 anonymous edits

TPT (Software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=444067274> *Contributors:* Andreas Kaufmann, Frozen4322, Iridescent, JaGa, Jens Lüdemann, Jluedem, Symplectic Map, Walter Görlitz, Xofpoa, 17 anonymous edits

Traceability matrix *Source:* <http://en.wikipedia.org/w/index.php?oldid=441935958> *Contributors:* Ahoerstemeier, Andreas Kaufmann, Charles Matthews, DRogers, Discospinster, Donmillion, Fry-kun, Furrykef, Graham87, Gurch, IPSOS, Kuru, Mdd, MrOllie, Pamar, Pinecar, Pravinparmarce, Rettetast, Ronz, Sardanaphalus, Shambhaviroy, Thebluemanager, Timmeu22, Voyagerfan5761, Walter Görlitz, WikiTome, 78 anonymous edits

Tree testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=358615870> *Contributors:* Andy Dingley, Dobrien, Jpbowen, PamD

TTCN-3 *Source:* <http://en.wikipedia.org/w/index.php?oldid=436205619> *Contributors:* Alan Liefing, Alexttcn, Alfe, Cfeet77, Erkilipre, JamesBWatson, R'n'B, 3 anonymous edits

Twist (software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=441001346> *Contributors:* De728631, Haakon, Ipsign, R'n'B, Teresa.ann.g

Unit testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=444598188> *Contributors:* .digamma, Ahc, Ahoerstemeier, AliveFreeHappy, Allan McInnes, Allen Moore, Anderbubble, Andreas Kaufmann, Andy Dingley, Angadn, Anorhup, Ardonik, Asavoia, Attilios, Autarch, Bakersg13, Bdiijkstra, BenFrantzDale, Brian Geppert, CanisRufus, Canterbury Tail, Chris Pickett, ChristianEdwardGruber, ChuckEsterbrook, Clausen, Colonies Chris, Corvi, Craigwb, DRogers, DanMS, Derbeth, Dillard421, Discospinster, Dmuler, Earlypsychosis, Edaelon, Edward Z. Yang, Eewild, El T, Etilio, Evil saline, Excerial, FlashSheridan, FrankTobia, Fredrik, Furrykef, GTBacchus, Ggggedxn, Goswaamivijay, Guille.boards, Haakon, Hanacy, Hari Surendran, Hayne, Hfastedge, Hooperbloob, Hsingh77, Hypersonic12, Ibbn, Influent1, J.delanoy, JamesBWatson, Jjamison, Joeggi, Jogloran, Jonhanson, Jpalm 98, Kamots, KaragouniS, Karl Dickman, Kku, Konman72, Kuru, Longhorn72, Looxix, Martin Majlis, Martinig, MaxHund, MaxSem, Mesece, Mheusser, Mhhanley, Michig, MickeyWiki, Miker@sundialservices.com, Mild Bill Hiccup, Mortense, Mr. Disguise, MrOllie, Mtoomezak, Nat hillary, Nate Silva, Nbryant, Neile, Nick Lewis CNH, Notinasnoid, Ohnoitsjamie, OmriSegal, Ottawa4ever, PGWG, Pabllasso, Paling Alchemist, Pantosys, Paul August, Paulocheque, Peb21, Pinecar, Pmerson, Radagast3, RainbowOfLight, Ravialluru, Ravindrat, RenniePet, Rich Farmbrough, Richardkmiller, Rjwilmsi, Rogerborg, Rookkey, RoyOsherove, Ryans.ryu, S.K., S3000, SAE1962, Saalam123, Shyam 48, SimonTrew, Sketch051, Sliigocki, Smalljim, So God created Manchester, Solde, Sozin, Ssd, Spiro, Stephen, SteveLoughran, Stumps, Sujith.srao, SVK, Swtchwr, Sybersnake, TFriesen, Themilofkeytone, Thv, Timo Honkasalo, Tiroche, Tobias Bergemann, Toddst1, Tony Morris, Tyler Oderkirk, Unittester123, User77764, VMS Mosaic, Veghead, Vishnava, Walter Görlitz, Willem-Paul, Winhunter, Wmahan, Zed toocool, 450 anonymous edits

Unusual software bug *Source:* <http://en.wikipedia.org/w/index.php?oldid=443967336> *Contributors:* 5 albert square, Aaronbrick, Abu badali, Acdx, Aeolian145, Ahaig, Ais523, AliveFreeHappy, Alksub, Andiiihiiii, AndreaPersphone, Andreas Kaufmann, Antonielly, Apankrat, Asmeurer, BWaldKqIer, Balabiot, Barrymyles, Beland, Bfg, Boing! said Zebedee, Booyabazooka, Branger, Bryan Derksen, Bumbulski, Canadian Monkey, Cavac, Chaos5023, Chrismacgr, Cosini Zeno, CyberShadow, Cybercobra, Czambra, Dannythorpe, Dar7yl, Darekun, David R. Ingham, DenisHowe, Denisarona, Eaglizard, Eastlaw, Elektrik Shoos, ERKURITA, Erkan Yilmaz, Esrob, Eyu100, Furrykef, Fuzlogic, Gnp, Hairy Dude, Happygiraffe, HarmonicFeather, Hbf, Holme053, Hooperbloob, Hope.forever.ends, Hydrargyrum, Jdiemer, Jmlptzlp, Joelblakesley, Kaagle, Keenan Pepper, Ketiltrout, LinguistAtLarge, Lionel Elie Mamane, Mckaysalsbury, Meand, Michael Hardy, Mindmatrix, MortenJ1638, MrH, Murray Langton, NJGW, Nataxia, Noosphere, Nycmstr, OranL, Ospalh, PhilKnight, Pichpich, Pinecar, Pinkadelica, Prgmr@wrk, Pseudomonas, Ptrb, RaptorHunter, Rhebus, RobinHood70, Rrostrom, Segv11, SethTisue, Sigfpe, Sintaku, Spyforthemoon, Steven Weston, Tedder, Thumperward, Tigrisek, Timwi, Trivialist, Uucp, Walter Görlitz, Wizard191, 101 anonymous edits

Usability testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=446191195> *Contributors:* 137.28.191.xxx, Aapo Laitinen, Al Tereego, Alan Pascoe, Alvin-cs, Antarkisawan, Arthena, Azrael81, Bihero, Bkillam, Brandon, Breakthru10technologies, Bretclement, ChrisJMoore, Christopher Agnew, Cjohansen, Kcatz, Conversion script, Crònica, DXBari, Dennis G. Jerz, Dickohead, Diego Moya, Dobrien, DrJohnBrooke, Dvandersluijs, EagleFan, Farreaching, Fredcondo, Geoffsaue, Gmarinp, Gokusandwich, GraemeL, Gubernet, Gumad, Hede2000, Hooperbloob, Hstetter, JDBravo, JaGa, Jean-Frédéric, Jetuusp, Jhouckwh, Jmike80, Karl smith, Kolyma, Kuru, Lakeworks, Leonard*Bloom, LizardWizard, Mandalaz, Manika, MaxHund, Mchali, Miamiichic, Michael Hardy, MichaelMcGuffin, MikeBlockQuickBooksCPA, Millahna, Mindmatrix, Omegatron, Pavel Vozenilek, Philippumd, Pigsonthewing, Pindakaas, Pinecar, QualMod, Ravialluru, Researcher1999, Rich Farmbrough, Rlsheehan, Ronz, Rossami, Schmettow, Shadowjays, Siddhi, Spalding, Tamarkot, Technopat, Tobias Bergemann, Toghme, Tomhab, Toomuchwork, Vmahi9, Wikinstone, Wikitonic, Willem-Paul, Wwheeler, Yettie0711, ZeroOne, 129 anonymous edits

Utest *Source:* <http://en.wikipedia.org/w/index.php?oldid=421454504> *Contributors:* Airplaneman, Alan Liefing, Baxtersmalls, ClamDip, Dmbius, Fabrictramp, Giraffedata, Gkapor16, Ibella88, J36miles, MrOllie, Ricky81682, Skier Dude, Tedickey, Woohookitty, 7 anonymous edits

Verification and Validation (software) *Source:* <http://en.wikipedia.org/w/index.php?oldid=443606287> *Contributors:* Akp1964, Alaerts, Alfie66, Altenmann, Andreas Kaufmann, BenAveling, CoderGnome, El Pantera, Embtech, Ensign bedrill, Fatespaces, GABaker, Gazpacho, GoetzUM, Gunnala, HeckXX, Henri662, Hu12, JONYKNUK, Jason Hampson, Jpbowen, KnowledgeOfSelf, Kosaraju007, LarsHolmberg, Lastorset, Lordmetroid, MER-C, Michig, Mr pand, Mskeel, Pgr94, Phansen, Pinecar, Raleign9, Rjwilmsi, Rlsheehan, Suk 9999, Superbecat, Suzanneb, Swiftblade21, Trakon, Walter Görlitz, XavierAJones, 124, נבו, נבו, 124 anonymous edits

Volume testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=364552806> *Contributors:* Closedmouth, EagleFan, Faught, Kumar74, Octahedron80, Pinecar, Terry1944, Thingg, Thru the night, Walter Görlitz, 7 anonymous edits

Vulnerability (computing) *Source:* <http://en.wikipedia.org/w/index.php?oldid=445075381> *Contributors:* Abune, Adequate, Ahouseholder, AnOddName, Anant k, Ash, BaldPark, Beetstra, Beland, Bentisa, Brownh2o, Cenzic, CesarB, Chris palmer, Chris the speller, CliffC, DARTH SIDIOUS 2, DanielPharos, Dcampbell30, Derek farn, Djptechie, Dman727, Dreftymac, Dtang2, Ehheh, ElKevo, Eleschinski2000, Emilyisdistinct, Erik9, Eugene van der Pijll, Fathisules, Fleetflame, FlyHigh, FrozenUmbrella, Gardar Rurak, Gruffi, Guriaz, HamburgerRadio, Haseo9999, Icetea8, InShaneev, Irishguy, J23450N, Jbolden1517, JonHarder, Joy, Jramsey, Jruderman, JuTiLi, Lambiam, Larry Yuma, Liquified, Logical Cowboy, Mandarax, Mani1, Manuc66, Michaeldsuarez, Mindmatrix, MistyHora, Mmemex, Mozzerati, Mugunth Kumar, Naraht, Neuffer, Pastore Italy, Persian Poet Gal, PotentialDanger, Pradameinhoff, Ptrb, Quarl, Rjwilmsi, Ronz, Ruid Koot, S.C.F, Sarveshbhatija, Sassy410, Sdfisher, Securitypbreaks, Sensiblekid, Shajure, Ssofttest123, Solarapex, Sweerek, Swwiki, Tanjstaffil, The Evil IP address, Thoo789789, Touisiau, Tslocum, VernoWhitney, Waldo, WalterGR, WhiteDragon, Wmahan, Xandi, Xezbeth, Zhenqinli, ZimZalaBim, 98 anonymous edits

Web testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=436663625> *Contributors:* Andreas Kaufmann, Cbuckley, Danielcornell, Darth Panda, Dhiraj1984, DthomasJL, JASpencer, JamesBWatson, Jettfreeman, Jwoodger, KarlDubost, MER-C, Macrofiend, Narayanraman, P199, Pinecar, Rchandra, Runnerweb, SEWilco, Softtest123, Testgeek, Thadius856, TubularWorld,

Walter Görlitz, 32 anonymous edits

White-box testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=446223542> *Contributors:* Ag2402, Aillema, AnOddName, Andreas Kaufmann, Arthena, Bobogoobo, CSZero, Caesura, Chris Pickett, Chrys, Closedmouth, Culix, DRogers, DanDoughty, DeadEyeArrow, Denisarona, Dupz, Ebde, Erkan Yilmaz, ErrOneous, Faught, Furrykef, Hooperbloob, Hu12, Hyad, Hyenaste, Isnow, Ixfd64, JStewart, JYolkowski, Jacksprat, Johntex, Jpalm 98, Juanmamb, Kanigan, Kasukurthi.vrc, Kuru, Mark.murphy, Mathieu, MaxDel, Menthaxpiperita, Mentifisto, Mezod, Michaeldunn123, Michig, Moeron, Mpilaeten, Mr Minchin, MrOllie, Noisy, Noot al-ghoubain, Nvrijn, Old Moonraker, PankajPeriwal, Philip Trueman, Pinecar, Pradameinhoff, Qxz, Radiojon, Ravialluru, Rsutherland, S.K., Solde, Suffusion of Yellow, Sushiflinger, Sven Manguard, Tedickey, The Rambling Man, Toddst1, Vellela, Walter Görlitz, Yady, Yilloslime, 114 anonymous edits

Windmill (testing framework) *Source:* <http://en.wikipedia.org/w/index.php?oldid=355439606> *Contributors:* AlistairMcMillan, Durova, Fleegix, Jclemens, Malcolmx15

X-Machine Testing *Source:* <http://en.wikipedia.org/w/index.php?oldid=422563014> *Contributors:* AJHSimons, Brevan, Foobarnix, Greenrd, Iridescent, Mike.stannett, 6 anonymous edits

Image Sources, Licenses and Contributors

Image:IBM360-65-1.corestore.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:IBM360-65-1.corestore.jpg> *License:* GNU Free Documentation License *Contributors:* Original uploader was ArnoldReinhold at en.wikipedia

Image:H96566k.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:H96566k.jpg> *License:* Public Domain *Contributors:* Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., 1988.

File:US Navy 090407-N-4669J-042 Sailors assigned to the air department of the aircraft carrier USS George H.W. Bush (CVN 77) test the ship's catapult systems during acceptance trials.jpg *Source:* [http://en.wikipedia.org/w/index.php?title=File:US_Navy_090407-N-4669J-042_Sailors_assigned_to_the_air_department_of_the_aircraft_carrier_USS_George_H.W._Bush_\(CVN_77\)_test_the_ship's_catapult_systems.d](http://en.wikipedia.org/w/index.php?title=File:US_Navy_090407-N-4669J-042_Sailors_assigned_to_the_air_department_of_the_aircraft_carrier_USS_George_H.W._Bush_(CVN_77)_test_the_ship's_catapult_systems.d) *License:* Public Domain *Contributors:*

File:Blackbox.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Blackbox.svg> *License:* Public Domain *Contributors:* Original uploader was Frap at en.wikipedia

Image:Codenomicon-logo-and-type2.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Codenomicon-logo-and-type2.png> *License:* Creative Commons Attribution 3.0 *Contributors:* Ari.takanen

Image:Core_primary_logo.PNG *Source:* http://en.wikipedia.org/w/index.php?title=File:Core_primary_logo.PNG *License:* Fair Use *Contributors:* Lamro

Image:DeviceAnywhere.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:DeviceAnywhere.jpg> *License:* Free Art License *Contributors:* Huunex

Image:Endeavour-logo-white.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Endeavour-logo-white.png> *License:* Public Domain *Contributors:* E-cuellar

Image:Endeavour-homepage.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Endeavour-homepage.png> *License:* GNU General Public License *Contributors:* Endeavour-mgmt, Tony Wills

Image:Endeavour-gantt.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Endeavour-gantt.png> *License:* GNU General Public License *Contributors:* Endeavour-mgmt, Tony Wills

Image:Fagan Inspection Simple flow.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Fagan_Inspection_Simple_flow.gif *License:* Public Domain *Contributors:* Monkeybait, Okok, 1 anonymous edits

Image:LayersUsers28 10 08.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:LayersUsers28_10_08.jpg *License:* Public Domain *Contributors:* Gderazon

Image:AutomationComponents3 29 10 08.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:AutomationComponents3_29_10_08.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Gderazon

Image:LayersUsersAppServ28 10 08.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:LayersUsersAppServ28_10_08.jpg *License:* Public Domain *Contributors:* Gderazon

Image:Msf3-hashdump small.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Msf3-hashdump_small.jpg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Self created session

Image:mbt-overview.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Mbt-overview.png> *License:* Public Domain *Contributors:* Antti.huima, Monkeybait

Image:mbt-process-example.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Mbt-process-example.png> *License:* Public Domain *Contributors:* Antti.huima, Monkeybait

File:Original Software logo.gif *Source:* http://en.wikipedia.org/w/index.php?title=File:Original_Software_logo.gif *License:* Fair Use *Contributors:* Muhandes

File:SoftwareQualityCharacteristicAttributeRelationship.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:SoftwareQualityCharacteristicAttributeRelationship.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Mbarberony

Image:RST Process.JPG *Source:* http://en.wikipedia.org/w/index.php?title=File:RST_Process.JPG *License:* Public Domain *Contributors:* Olga.Petrus

Image:Sputnik-test-icon.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:Sputnik-test-icon.gif> *License:* Fair Use *Contributors:* Daniel Hen, Salavat, Terrillja

Image:Test-driven development.PNG *Source:* http://en.wikipedia.org/w/index.php?title=File:Test-driven_development.PNG *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Excirial (Contact me, Contribs)

File:TOSCA_Logo.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:TOSCA_Logo.jpg *License:* Fair Use *Contributors:* TRICENTIS Technology & Consulting GmbH

Image:TPT Logo.png *Source:* http://en.wikipedia.org/w/index.php?title=File:TPT_Logo.png *License:* unknown *Contributors:* Jluedem

Image:TPT Screenshot.png *Source:* http://en.wikipedia.org/w/index.php?title=File:TPT_Screenshot.png *License:* Creative Commons Attribution 3.0 *Contributors:* Jluedem

Image:Virzis Formula.PNG *Source:* http://en.wikipedia.org/w/index.php?title=File:Virzis_Formula.PNG *License:* Public Domain *Contributors:* Original uploader was Schmettow at en.wikipedia. Later version(s) were uploaded by NickVeys at en.wikipedia.

Image:UTest company logo.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:UTest_company_logo.jpg *License:* Fair Use *Contributors:* Baxtersmall

File:2010-T10-ArchitectureDiagram.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:2010-T10-ArchitectureDiagram.png> *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Neil Smithline

Image:Wm_logo_round.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Wm_logo_round.png *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* Fleggix

License

Creative Commons Attribution-Share Alike 3.0 Unported
<http://creativecommons.org/licenses/by-sa/3.0/>
