Software testing

PDF generated using the open source mwlib toolkit. See http://code.pediapress.com/ for more information.
PDF generated at: Wed, 24 Aug 2011 09:57:44 UTC

Contents

Articles

Introduction	**1**
Software testing	1
Black-box testing	**16**
Black-box testing	16
Exploratory testing	17
San Francisco depot	19
Session-based testing	20
Scenario testing	22
Equivalence partitioning	22
Boundary-value analysis	24
All-pairs testing	25
Fuzz testing	26
Cause-effect graph	30
Model-based testing	31
Web testing	35
Installation testing	37
White-box testing	**38**
White-box testing	38
Code coverage	39
Modified Condition/Decision Coverage	44
Fault injection	45
Bebugging	50
Mutation testing	51
Testing of non functional software aspects	**55**
Non-functional testing	55
Software performance testing	56
Stress testing	62
Load testing	63
Volume testing	67
Scalability testing	67
Compatibility testing	68

Portability testing 69
Security testing 69
Attack patterns 71
Localization testing 75
Pseudolocalization 76
Recovery testing 77
Soak testing 78
Characterization test 78

Unit testing **80**

Unit testing 80
Self-testing code 85
Test fixture 86
Method stub 88
Mock object 89
Lazy systematic unit testing 92
Test Anything Protocol 93
xUnit 96
List of unit testing frameworks 98
SUnit 129
JUnit 130
CppUnit 132
Test::More 133
NUnit 134
NUnitAsp 136
csUnit 138
HtmlUnit 140

Test automation **141**

Test automation 141
Test bench 145
Test execution engine 146
Test stubs 148
Testware 149
Test automation framework 150
Data-driven testing 151
Modularity-driven testing 152
Keyword-driven testing 152

Hybrid testing 154
Lightweight software test automation 155

Testing process 156

Software testing controversies 156
Test-driven development 158
Agile testing 165
Bug bash 166
Pair Testing 166
Manual testing 167
Regression testing 169
Ad hoc testing 171
Sanity testing 171
Integration testing 173
System testing 174
System integration testing 176
Acceptance testing 178
Risk-based testing 182
Software testing outsourcing 183
Tester driven development 185
Test effort 185

Testing artefacts 187

IEEE 829 187
Test strategy 189
Test plan 192
Traceability matrix 194
Test case 195
Test data 197
Test suite 198
Test script 199
Test harness 200

Static testing 201

Static testing 201
Software review 202
Software peer review 204
Software audit review 205

Software technical review 206
Management review 207
Software inspection 208
Fagan inspection 210
Software walkthrough 213
Code review 214
Automated code review 216
Code reviewing software 217
Static code analysis 218
List of tools for static code analysis 220

GUI testing and review **226**

GUI software testing 226
Usability testing 229
Think aloud protocol 234
Usability inspection 235
Cognitive walkthrough 235
Heuristic evaluation 238
Pluralistic walkthrough 241
Comparison of usability evaluation methods 244

References

Article Sources and Contributors 246
Image Sources, Licenses and Contributors 252

Article Licenses

License 253

Introduction

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 programing 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.ambysoft.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-)

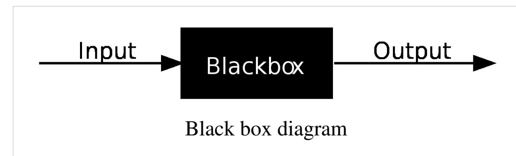
[58] Microsoft Development Network Discussion on exactly this topic (http://channel9.msdn.com/forums/Coffeehouse/402611-Are-you-a-Test-Driven-Developer/)

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)

Black-box testing

Black-box testing



Black box diagram

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), Working Draft 3.4, 27. April 2001.

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 Perlclip.

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/QAIExploring.pdf), p. 36.
[3] Cem Kaner, *A Tutorial in Exploratory Testing* (http://www.kaner.com/pdfs/QAIExploring.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. p. 7. . Retrieved October 23, 2010.
[8] Kaner, Cem (2008). "A Tutorial in Exploratory Testing" (http://www.kaner.com/pdfs/QAIExploring.pdf). 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)

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

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 aconymn 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/)

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.

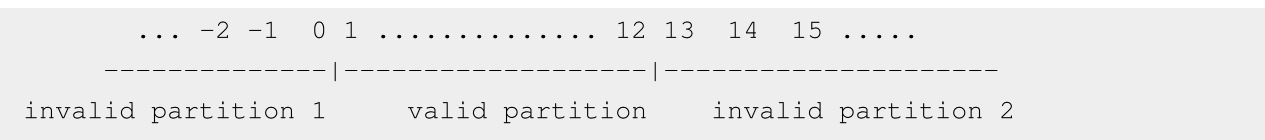
External links

- Introduction to Scenario Testing (http://www.kaner.com/pdfs/ScenarioIntroVer4.pdf)

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.



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

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:

```
        ... -2 -1  0 1 ..... 6 7 ..... 12 13  14  15 .....
      --------------|---------|----------|---------------------
 invalid partition 1      P1         P2     invalid partition 2
                        valid partitions
```

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 [2]
- Parteg [3], 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://www.testingstandards.co.uk

[3] http://parteg.sourceforge.net

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<=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 [2] website.

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)

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 API's.[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 Takanen, 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)

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.

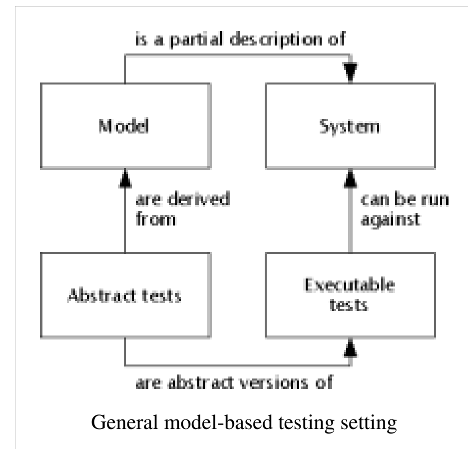
A cause-effect graph is useful for generating a reduced decision table.

Further reading

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

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).



General model-based testing setting

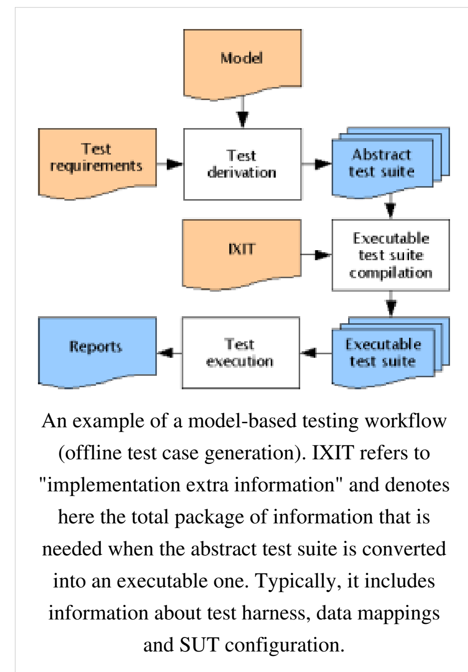
The model describing the SUD is usually an abstract, partial presentation of the system under test's desired behavior. The test cases derived from this model are functional tests on the same level of abstraction as the model. These test cases are collectively known as the abstract test suite. The abstract 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



An example of a model-based testing workflow (offline test case generation). IXIT refers to "implementation extra information" and denotes here the total package of information that is needed when the abstract test suite is converted into an executable one. Typically, it includes information about test harness, data mappings and SUT configuration.

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 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/%7Eatif/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=ppmmedialaboutThisBook) 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.

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.

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”.

White-box testing

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

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 satisifed. 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=jykvlTCoksMC&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)

[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/)

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; Rierson, 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

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.

- Corruption 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/)

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.

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

Testing of non functional software aspects

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

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

Stress testing

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.

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 or 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

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.

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)

Further reading

Designing Distributed Applications with Visual Studio .NET: Scalability (http://msdn2.microsoft.com/en-us/library/aa292172(VS.71).aspx)

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

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. Result are expressed in terms of the time required to move the software and complete data conversion and documentation updates.

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)

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
- Hoglund, 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. Dobb'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)

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.

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:

```
[!!! εÐiţ Þr0ǵЯãm səTτıИğ§ !!!]
```

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

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.

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.

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.

Unit testing

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.

```java
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.

```java
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)

Self-testing code

Self-testing code is software which incorporates built-in tests (see test-first development).

In Java, to execute a unit test from the command line, a class can have methods like the following.

```
// Executing <code>main</code> runs the unit test.
public static void main(String[] args) {
    test();
}

static void test() {
    assert foo == bar;
}
```

To invoke a full system test, a class can incorporate a method call.

```
public static void main(String[] args) {
    test();
    TestSuite.test();    // invokes full system test
}
```

Test fixture

A **test fixture** is something used to consistently test some item, device, or piece of software.

Electronics

Circuit boards, electronic components, and chips are held in place and subjected to controlled electronic test signals. One example is a bed of nails tester.

Software

Test fixture refers to the fixed state used as a baseline for running tests in software testing. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. Some people call this the *test context*.

Examples of fixtures:

- Loading a database with a specific, known set of data
- Erasing a hard disk and installing a known clean operating system installation
- Copying a specific known set of files
- Preparation of input data and set-up/creation of fake or mock objects

Test fixture in xUnit

In generic xUnit, a *test fixture* is all the things that must be in place in order to run a test and expect a particular outcome.

Frequently fixtures are created by handling *setUp()* and *tearDown()* events of the unit testing framework. In *setUp()* one would create the expected state for the test, and in *tearDown()* it would clean up what had been set up.

Four phases of a test:

1. **Set up** -- Setting up the *test fixture*.
2. **Exercise** -- Interact with the *system under test*.
3. **Verify** -- Determine whether the expected outcome has been obtained.
4. **Tear down** -- Tear down the *test fixture* to return to the original state.

Use of fixtures

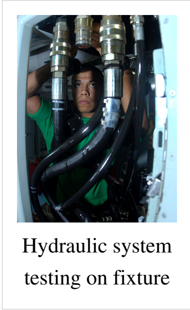
Some advantages of *fixtures* include separation of the test initialization (and destruction) from the testing, reusing a known state for more than one test, and special assumption by the testing framework that the fixture set up works.

Physical testing

In physical testing, a fixture is a device or apparatus to hold or support the test specimen during the test. The influence of test fixtures on test results is important and is an ongoing subject of research.[1]

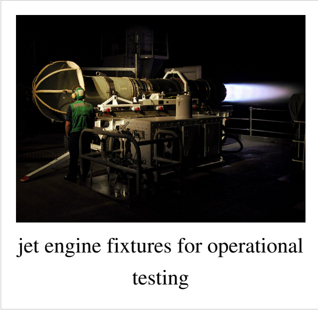
Many test methods detail the requirements of test fixtures in the text of the document.[2] [3]

Test fixture on universal testing machine for three point flex test



Hydraulic system testing on fixture

Test structure fixtures on large seismic table for earthquake tests



jet engine fixtures for operational testing

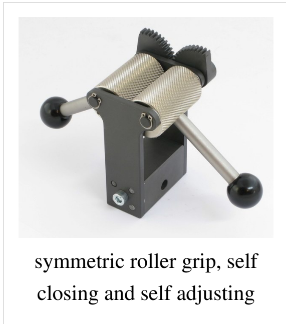
Some fixtures employ clamps, wedge grips and pincer grips.

pincer clamps max. 50kN spring-biased

offset compensated wedge grip max.50kN

different vice and screw grips of a German manufacturer

Further types of construction are eccentric roller fixtures, thread grips and button head grips as well as rope grips.



symmetric roller grip, self closing and self adjusting

multiple button head grip for speedy tests on series

small rope grip 200N to test fine wires

very compact wedge grip for temperature chambers providing extreme temperatures

Mechanical holding apparatus provide the clamping force via arms, wedges or eccentric wheel to the jaws. Additional there are pneumatic and hydraulic fixtures for tensile testing that do allow very fast clamping procedures and very high clamping forces

pneumatic grip, symmetrical, clamping force 2.4kN

heavy duty hydraulic clamps, clamping force 700kN

Bending device for tensile testing machines

Equipment to proof pealing forces up to 10kN

References

[1] Abadalah, MG; Gascoigne, HE (1989). *The Influence of Test Fixture Design on the Shear Test for Fiber Composite Materials*. ASTM STP.
[2] ASTM B829 Test for Determining the Formability of copper Strip
[3] ASTM D6641 Compressive Properties of Polymer Matrix Using a Combined Loading Compression Test Fixture

- http://www.cs.utep.edu/~cheon/cs3331/notes/unit-testing.ppt (http://www.cs.utep.edu/~cheon/cs3331/notes/unit-testing.ppt)
- http://manuals.rubyonrails.com/read/chapter/26

Method stub

A **method stub** or simply **stub** in software development is a piece of code used to stand in for some other programming functionality. A stub may simulate the behavior of existing code (such as a procedure on a remote machine) or be a temporary substitute for yet-to-be-developed code. Stubs are therefore most useful in porting, distributed computing as well as general software development and testing.

An example of a stub in pseudocode might be as follows:

```
BEGIN
    Temperature = ThermometerRead(Outside)
    IF Temperature > 40 THEN
         PRINT "It's HOT!"
    END IF
END
```

```
BEGIN ThermometerRead(Source insideOrOutside)
     RETURN 28
END ThermometerRead
```

The above pseudocode utilises the function `ThermometerRead`, which returns a temperature. While `ThermometerRead` would be intended to read some hardware device, this function currently does not contain the necessary code. So `ThermometerRead` does not, in essence, simulate any process, yet it *does* return a legal value, allowing the main program to be at least partly tested. Also note that although it accepts the parameter of type `Source`, which determines whether inside or outside temperature is needed, it does not use the actual value passed (argument `insideOrOutside`) by the caller in its logic.

A stub[1] is a routine that doesn't actually do anything other than declare itself and the parameters it accepts and returns something that is usually the values expected in one of the "happy scenarios" for the caller. Stubs are used commonly as placeholders for implementation of a known interface, where the interface is finalized/known but the implementation is not yet known/finalized. The stub contains just enough code to allow it to be compiled and linked with the rest of the program.

References

[1] http://www.webopedia.com/ (http://www.webopedia.com/TERM/S/stub.html)

External links

- A Stub Generation System For C++ (http://www.cs.ncl.ac.uk/research/pubs/articles/papers/94.pdf) (PDF)
- Stub/mock frameworks for Java (http://www.sizovpoint.com/2009/03/java-mock-frameworks-comparison.html) Review and comparison of stub & mock frameworks for Java

Mock object

In object-oriented programming, **mock objects** are simulated objects that mimic the behavior of real objects in controlled ways. A computer programmer typically creates a mock object to test the behavior of some other object, in much the same way that a car designer uses a crash test dummy to simulate the dynamic behavior of a human in vehicle impacts.

Reasons for use

In a unit test, mock objects can simulate the behavior of complex, real (non-mock) objects and are therefore useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:

- supplies non-deterministic results (e.g. the current time or the current temperature);
- has states that are difficult to create or reproduce (e.g. a network error);
- is slow (e.g. a complete database, which would have to be initialized before the test);
- does not yet exist or may change behavior;
- would have to include information and methods exclusively for testing purposes (and not for its actual task).

For example, an alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the test must wait until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

Technical details

Mock objects have the same interface as the real objects they mimic, allowing a client object to remain unaware of whether it is using a real object or a mock object. Many available mock object frameworks allow the programmer to specify which, and in what order, methods will be invoked on a mock object and what parameters will be passed to them, as well as what values will be returned. Thus, the behavior of a complex object such as a network socket can be mimicked by a mock object, allowing the programmer to discover whether the object being tested responds appropriately to the wide variety of states such objects may be in.

Mocks, fakes and stubs

Some authors[1] draw a distinction between *fake* and *mock* objects. Fakes are the simpler of the two, simply implementing the same interface as the object that they represent and returning pre-arranged responses. Thus a fake object merely provides a set of method stubs.

In the book "The Art of Unit Testing"[2] mocks are described as a fake object that helps decide if a test failed or passed, by verifying if an interaction on an object occurred or not. Everything else is defined as a stub. In that book, "Fakes" are anything that is not real. Based on their usage, they are either stubs or mocks.

Mock objects in this sense do a little more: their method implementations contain assertions of their own. This means that a true mock, in this sense, will examine the context of each call— perhaps checking the order in which its methods are called, perhaps performing tests on the data passed into the method calls as arguments.

Setting expectations

Consider an example where an authorization sub-system has been mocked. The mock object implements an `isUserAllowed(task : Task) : boolean`[3] method to match that in the real authorization class. Many advantages follow if it also exposes an `isAllowed : boolean` property, which is not present in the real class. This allows test code easily to set the expectation that a user will, or will not, be granted permission in the next call and therefore readily to test the behavior of the rest of the system in either case.

Similarly, a mock-only setting could ensure that subsequent calls to the sub-system will cause it to throw an exception, or hang without responding, or return `null` etc. Thus it is possible to develop and test client behaviors for all realistic fault conditions in back-end sub-systems as well as for their expected responses. Without such a simple and flexible mock system, testing each of these situations may be too laborious for them to be given proper consideration.

Writing log strings

A mock database object's `save(person : Person)` method may not contain much (if any) implementation code. It might or might not check the existence and perhaps the validity of the Person object passed in for saving (see fake vs. mock discussion above), but beyond that there might be no other implementation.

This is a missed opportunity. The mock method could add an entry to a public log string. The entry need be no more than "Person saved",[4] :146–7 or it may include some details from the person object instance, such as a name or ID. If the test code also checks the final contents of the log string after various series of operations involving the mock database then it is possible to verify that in each case exactly the expected number of database saves have been performed. This can find otherwise invisible performance-sapping bugs, for example, where a developer, nervous of losing data, has coded repeated calls to `save()` where just one would have sufficed.

Use in test-driven development

Programmers working with the test-driven development (TDD) method make use of mock objects when writing software. Mock objects meet the interface requirements of, and stand in for, more complex real ones; thus they allow programmers to write and unit-test functionality in one area without actually calling complex underlying or collaborating classes.[4] :144–5 Using mock objects allows developers to focus their tests on the behavior of the system under test (SUT) without worrying about its dependencies. For example, testing a complex algorithm based on multiple objects being in particular states can be clearly expressed using mock objects in place of real objects.

Apart from complexity issues and the benefits gained from this separation of concerns, there are practical speed issues involved. Developing a realistic piece of software using TDD may easily involve several hundred unit tests. If many of these induce communication with databases, web services and other out-of-process or networked systems, then the suite of unit tests will quickly become too slow to be run regularly. This in turn leads to bad habits and a

reluctance by the developer to maintain the basic tenets of TDD.

When mock objects are replaced by real ones then the end-to-end functionality will need further testing. These will be integration tests rather than unit tests.

Limitations

The use of mock objects can closely couple the unit tests to the actual implementation of the code that is being tested. For example, many mock object frameworks allow the developer to specify the order of and number of times that the methods on a mock object are invoked; subsequent refactoring of the code that is being tested could therefore cause the test to fail even though the method still obeys the contract of the previous implementation. This illustrates that unit tests should test a method's external behavior rather than its internal implementation. Over-use of mock objects as part of a suite of unit tests can result in a dramatic increase in the amount of maintenance that needs to be performed on the tests themselves during system evolution as refactoring takes place. The improper maintenance of such tests during evolution could allow bugs to be missed that would otherwise be caught by unit tests that use instances of real classes. Conversely, simply mocking one method might require far less configuration than setting up an entire real class and therefore reduce maintenance needs.

Mock objects have to accurately model the behavior of the object they are mocking, which can be difficult to achieve if the object being mocked comes from another developer or project or if it has not even been written yet. If the behavior is not modeled correctly then the unit tests may register a pass even though a failure would occur at run time under the same conditions that the unit test is exercising, thus rendering the unit test inaccurate.[5]

References

[1] Feathers, Michael (2005). "Sensing and separation". *Working effectively with legacy code*. NJ: Prentice Hall. p. 23 et seq. ISBN 0-13-117705-2.

[2] Osherove, Roy (2009). "Interaction testing with mock objects et seq". *The art of unit testing*. Manning. ISBN 978-1933988276.

[3] These examples use a nomenclature that is similar to that used in Unified Modeling Language

[4] Beck, Kent (2003). *Test-Driven Development By Example*. Boston: Addison Wesley. ISBN 0-321-14653-0.

[5] InJava.com (http://www.onjava.com/pub/a/onjava/2004/02/11/mocks.html#Approaches) to Mocking | O'Reilly Media

External links

- Tim Mackinnon (8 September 2009). "A Brief History of Mock Objects" (http://www.mockobjects.com/2009/09/brief-history-of-mock-objects.html). Mockobjects.com/.
- The Art of Unit Testing (two free PDF chapters and lots of videos) (http://ArtOfUnitTesting.com)
- Interaction Testing with the Typemock Isolator Mocking framework (http://typemock.org/getting-started-step-1-set/)
- Great Java mock frameworks comparison article: Java mock framework comparison (http://www.sizovpoint.com/2009/03/java-mock-frameworks-comparison.html)
- Test Doubles (http://xunitpatterns.com/Test Double.html): a section of a book on unit testing patterns.
- All about mock objects! Portal concerning mock objects (http://www.mockobjects.com)
- Mock Roles, not Objects (http://www.jmock.org/oopsla2004.pdf), a paper on the technique that was presented at OOPSLA 2004.
- Using mock objects for complex unit tests (http://www-128.ibm.com/developerworks/rational/library/oct06/pollice/index.html) IBM developerWorks
- Unit testing with mock objects (http://www.ibm.com/developerworks/java/library/j-mocktest/index.html) IBM developerWorks
- Using Mock Objects with Test Driven Development (http://www.theserverside.com/tt/articles/article.tss?l=JMockTestDrivenDev)

- Mock Object Patterns at Hillside (http://hillside.net/plop/plop2003/Papers/Brown-mock-objects.pdf) Mock Object Design Patterns
- Mocks Aren't Stubs (http://martinfowler.com/articles/mocksArentStubs.html) (Martin Fowler) Article about developing tests with Mock objects. Identifies and compares the "classical" and "mockist" schools of testing. Touches on points about the impact on design and maintenance.
- Mocking the Embedded World (http://www.atomicobject.com/pages/Embedded+Software#MockingEmbeddedWorld) Paper and sample project concerned with adapting mocking and Presenter First for embedded software development.
- Surviving Mock Abuse (http://www.ibm.com/developerworks/library/j-mocktest.html) Pitfalls of overuse of mocks and advice for avoiding them
- Responsibility Driven Design with Mock Objects (http://www.methodsandtools.com/archive/archive.php?id=90)
- The Art of Unit Testing in Java (http://manning.com/koskela2/)
- Mock framework for Microsoft Dynamics AX 2009 (http://axmocks.codeplex.com/)
- Interaction Based Testing with Rhino Mocks (http://www.testingtv.com/2009/08/28/interaction-based-testing-with-rhino-mocks/)
- Unit Testing with Mock Objects via MockBox (http://blog.coldbox.org/post.cfm/unit-testing-with-mock-objects-amp-mockbox)
- Mockups (http://www.mockuptiger.com/mockups) Low Fidelity mockups for UI design

Lazy systematic unit testing

Lazy Systematic Unit Testing[1] is a software unit testing method based on the two notions of *lazy specification*, the ability to infer the evolving specification of a unit on-the-fly by dynamic analysis, and *systematic testing*, the ability to explore and test the unit's state space exhaustively to bounded depths. A testing toolkit JWalk exists to support lazy systematic unit testing in the Java programming language[2] .

Lazy Specification

Lazy specification refers to a flexible approach to software specification, in which a specification evolves rapidly in parallel with frequently modified code[1] . The specification is inferred by a semi-automatic analysis of a prototype software unit. This can include static analysis (of the unit's interface) and dynamic analysis (of the unit's behaviour). The dynamic analysis is usually supplemented by limited interaction with the programmer.

The term **Lazy specification** is coined by analogy with *lazy evaluation* in functional programming. The latter describes the delayed evaluation of sub-expressions, which are only evaluated on demand. The analogy is with the late stabilization of the specification, which evolves in parallel with the changing code, until this is deemed stable.

Systematic Testing

Systematic testing refers to a complete, conformance testing approach to software testing, in which the tested unit is shown to conform exhaustively to a specification, up to the testing assumptions[3] . This contrasts with exploratory, incomplete or random forms of testing. The aim is to provide repeatable guarantees of correctness after testing is finished.

Examples of systematic testing methods include the Stream X-Machine testing method[4] and equivalence partition testing with full boundary value analysis.

References

[1] A J H Simons, JWalk: Lazy systematic unit testing of Java classes by design introspection and user interaction, *Automated Software Engineering, 14 (4), December*, ed. B. Nuseibeh, (Boston: Springer, 2007), 369-418.

[2] *The JWalk Home Page*, http://www.dcs.shef.ac.uk/~ajhs/jwalk/

[3] A J H Simons, A theory of regression testing for behaviourally compatible object types, *Software Testing, Verification and Reliability, 16 (3), UKTest 2005 Special Issue, September,* eds. M Woodward, P McMinn, M Holcombe and R Hierons (Chichester: John Wiley, 2006), 133-156.

[4] F Ipate and W M L Holcombe, Specification and testing using generalised machines: a presentation and a case study, *Software Testing, Verification and Reliability, 8 (2),* (Chichester: John Wiley, 1998), 61-81.

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.
- yuitest [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://yuilibrary.com/yuitest/

xUnit

Various code-driven testing frameworks have come to be known collectively as **xUnit**. These frameworks allow testing of different elements (units) of software, such as functions and classes. The main advantage of xUnit frameworks is that they provide an automated solution with no need to write the same tests many times, and no need to remember what should be the result of each test. Such frameworks are based on a design by Kent Beck, originally implemented for Smalltalk as SUnit. Erich Gamma and Kent Beck ported SUnit to Java, creating JUnit. From there, the framework was also ported to other languages, e.g., CppUnit (for C++), NUnit (for .NET). They are all referred to as **xUnit** and are usually free, open source software. They are now available for many programming languages and development platforms.

xUnit architecture

All xUnit frameworks share the following basic component architecture, with some varied implementation details.

Test case

This is the most elemental class. All unit tests are inherited from here.

Test fixtures

A test fixture (also known as a test context) is the set of preconditions or state needed to run a test. The developer should set up a known good state before the tests, and after the tests return to the original state.

Test suites

A test suite is a set of tests that all share the same fixture. The order of the tests shouldn't matter.

Test execution

The execution of an individual unit test proceeds as follows:

```
setup(); /* First, we should prepare our 'world' to make an isolated
environment for testing */
...
/* Body of test - Here we make all the tests */
...
teardown(); /* In the end, whether succeed or fail we should clean up
our 'world' to
not disturb other tests or code */
```

The setup() and teardown() methods serve to initialize and clean up test fixtures.

Assertions

An assertion is a function or macro that verifies the behavior (or the state) of the unit under test. Failure of an assertion typically throws an exception, aborting the execution of the current test.

xUnit Frameworks

Many xUnit frameworks exist for various programming languages and development platforms.

- List of unit testing frameworks

xUnit Extensions

Extensions are available to extend xUnit frameworks with additional specialized functionality. Examples of such extensions include XMLUnit [1], DbUnit [2], HtmlUnit and HttpUnit.

External links

- Kent Beck's original testing framework paper [3]
- Other list of various unit testing frameworks [4]
- OpenSourceTesting.org lists many unit testing frameworks, performance testing tools and other tools programmers/developers may find useful [5]
- Test automation patterns for writing tests/specs in xUnit. [6]
- Martin Fowler on the background of xUnit. [7]

References

[1] http://xmlunit.sourceforge.net/
[2] http://www.dbunit.org/
[3] http://www.xprogramming.com/testfram.htm
[4] http://www.xprogramming.com/software.htm
[5] http://opensourcetesting.org/
[6] http://xunitpatterns.com/
[7] http://www.martinfowler.com/bliki/Xunit.html

List of unit testing frameworks

This page is a list of tables of code-driven unit testing frameworks for various programming languages. Some but not all of these are based on xUnit.

Columns (Classification)

- **Name**: This column contains the name of the framework and will usually link to it.
- **xUnit**: This column indicates whether a framework should be considered of xUnit type.
- **TAP**: This column indicates whether a framework can emit TAP output for TAP-compliant testing harnesses.
- **Generators**: Indicates whether a framework supports data generators. Data generators generate input data for a test and the test is run for each input data that the generator produces.
- **Fixtures**: Indicates whether a framework supports test-local fixtures. Test-local fixtures ensure a specified environment for a single test.
- **Group fixtures**: Indicates whether a framework supports group fixtures. Group fixtures ensure a specified environment for a whole group of Tests
- **Other columns**: These columns indicate whether a specific language / tool feature is available / used by a framework.
- **Remarks**: Any remarks.

Languages

ABAP

Name	xUnit	Homepage	Remarks
ABAP Unit	Yes	[1]	since SAP NetWeaver 2004

ActionScript / Adobe Flex

Name	xUnit	Homepage	Remarks
FlexUnit		[2]	
FlexUnit 4	Yes	[3]	Metadata-driven unit testing for Flex 2,3 and 4 and ActionScript 3 projects
Reflex Unit		[4]	Metadata-driven unit testing framework for Flex 2 and 3
FUnit	Yes	[5]	Metadata-driven unit testing for Flex
ASTUce	Yes	[6]	Unit testing for ActionScript 3 (also JS, AS1, AS2), that can also run on the command-line with a cross-platform executable (support OS X / Linux / Windows)
AsUnit		[7]	Flash Players 6, 7, 8, 9 and 10
dpUInt		[8]	Unit and Integration testing framework for Flex 2 and 3
Fluint		[9]	Unit and Integration testing framework for Flex 2 and 3
mojotest	Yes	[10]	(under development) Unit testing for ActionScript 3, Flash Player 10

Ada

Name	xUnit	Homepage	Remarks
AUnit		[11]	
AdaTEST 95		[12]	
Ahven		[13]	
TBrun		[14]	
VectorCAST/Ada		[15]	

AppleScript

Name	xUnit	Homepage	Remarks
ASUnit	Yes	[16]	Testing framework for AppleScript, influenced by SUnit, ASTest and Python unittest
ASTest	Yes	[17]	A testing framework for AppleScript

ASP

Name	xUnit	Homepage	Remarks
ASPUnit		[18]	

BPEL

Name	xUnit	Homepage	Remarks
BPELUnit		[19]	

C

Name	xUnit	Fixtures	Group fixtures	Generators	Homepage	License	Remarks
AceUnit	Yes	Yes			[20]		AceUnit is JUnit 4.x style, easy, modular and flexible. AceUnit can be used in resource constraint environments, e.g. embedded software development, as well as on PCs, Workstations and Servers (Windows and UNIX).
API Sanity Autotest	Yes	Yes (spectypes)	Yes (spectypes)	Yes	[21]	LGPL	Unit test generator for C/C++ libraries. Can automatically generate reasonable input data for every API function.
Automated Testing Framework					[22]	BSD	Originally developed for the NetBSD operating system but works well in most Unix-like platforms. Ability to install tests as part of a release.
Autounit (GNU)					[23]	LGPL	In beta/under construction

C++test	Yes	Yes	Yes	Yes	[24]	Commercial	Automated software quality solution that includes unit test generation and execution as well as reporting industry standard code coverage.
Cantata++	No	Yes	Yes	Yes	[25]	Commercial	Automated unit and integration testing on host and embedded systems with code coverage and unique call interface control to simulate and intercept calls.
Catsrunner					[26]	GPL	nice unit testing framework for cross-platform embedded development
cfix	Yes				[27]		Specialized for Windows development—both Win32 and NT kernel mode. Compatible to WinUnit.
Cgreen					[28]	LGPL	includes mocks
Check	Yes				[29]	LGPL	
Cmockery	Yes				[30]	Apache License 2.0	Google sponsored project.
CU					[31]	LGPL	CU is a simple unit testing framework for handling automated tests in C.
CUnit	Yes				[32]	LGPL	OS Independent (Windows, Linux, Mac OS X and probably others)
CUnitWin32	Yes				[33]		For Win32. Minimalistic framework. Executes each test as a separate process.
CUT	No				[34]	BSD	
CuTest	Yes				[35]	zlib	Simple, straightforward, fast. Single .c file. Used in the Apache Portable Runtime Library.
Cutter	Yes				[36]	LGPL	A Unit Testing Framework for C.
EmbeddedUnit	Yes	Yes			[37]	MIT	Embedded C
FCTX	Yes				[38]	BSD	Fast and complete unit testing framework all in one header. Declare and write your functions in one step. No dependencies. Cross platform.
GLib Testing	Yes	Yes			[39]		Part of GLib
GUnit					[40]		for GNOME
LibU	Yes	No			[41]	BSD	multiplatform (UNIXes and Windows); explicit test case/suite dependencies; parallel and sandboxed execution; xml, txt and customizable report formatting.
MinUnit					[42]	as-is	extreme minimalist unit testing using 2 C macros
RCUNIT	Yes				[43]	GPL	A robust C unit testing framework
RTRT					[44]		
SeaTest	Yes	Yes			[45]	MIT	Simple, pure C, unit testing framework
Smarttester					[46]		Automated unit and integration testing, and code coverage
TBrun					[14]		Automated unit and integration testing, and code coverage

Tessy					[47]		Automated unit and integration testing, and code coverage
TestApe					[48]		Test and mocking framework. Automatic default mocks for unresolved externals
Test Dept.	Yes				[49]	GPL	Can modify calls from software under test; e.g. test error conditions by stubbing malloc and letting it return null. Well documented
TPT	Yes	Yes	Yes	Yes	[50]	Commercial	Time Partition Testing:Automated model based unit and integration testing for embedded systems.
Unity	Yes			Yes	[51]	MIT	Lightweight & includes features for embedded development. Can work with Mocks and Exceptions via CMock [52] and CException [53]. Also integrated with test build environment Ceedling [54].
VectorCAST/C					[15]		Automated unit and integration testing, and code coverage
Visual Assert	Yes				[55]		Unit-Testing Add-In for Visual Studio. Based on the cfix testing framework.
xTests					[56]	BSD	Depends on STLSoft C & C++ Libraries
LCUT	Yes	Yes	Yes		[57]	Apache License 2.0	a Lightweight C Unit Testing framework, including mock support

C#

See .NET Programming languages below.

C++

Name	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping	Homepage	Remarks
Aeryn	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[58]	
API Sanity Autotest	Yes	Yes (spectypes)	Yes (spectypes)	Yes						[21]	Unit test generator for C/C++ libraries. Can automatically generate reasonable input data for every API function. LGPL.
ATF						Yes	Yes	Yes	Yes	[22]	BSD Licensed. Originally developed for the NetBSD operating system but works well in most Unix-like platforms. Ability to install tests as part of a release.
Boost Test Library	No[59]	Yes[60]	Yes[61]	Yes	No	Yes	User decision	Yes	Suites	[62]	Part of Boost

C++test	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	[24]	Commercial. Automated software quality solution that includes unit test generation and execution as well as reporting industry standard code coverage.
Cantata++	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	[63]	Commercial. Automated OO unit and integration testing on host and embedded systems with code coverage and unique call interface control to simulate and intercept calls.
CATCH	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	[64]	Header only, no external dependencies, auto-registration, tdd and bdd features
cfix	Yes	Yes	No	No	No	Yes	Yes	No		[65]	Specialized for Windows development—both Win32 and NT kernel mode. Compatible to WinUnit.
Cput	Yes	Yes	Yes	Yes		Yes	Yes	No	Suites	[66]	Library and MS Visual Studio add-in to create and run unit tests. Open Source.
CppTest		Yes					Yes		Suites	[67]	Released under LGPL
CppUnit	Yes	Yes	Yes	No	No	Yes	Yes	No	Suites	[68]	
CppUTest	Yes	Yes	Yes	No	No	No	Yes	No	Suites	[69]	Limited C++ set by design to keep usage easy and allow it to work on embedded platforms. Ported to Symbian and IAR
CppUnitLite	Yes			No	No	No	Yes	No	Suites	[70]	
CPUnit	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	[71]	Released under BSD.

CUTE	Yes	Yes	No	No	No	Yes			Suites	[72]	CUTE (C++ Unit Testing Easier) with Eclipse CDT integration. Single line include, without inheritance.
cutee	No	No	No	Yes						[73]	
CxxTest	Yes	Yes	Yes	No	Yes*	Optional	Yes	No	Suites	[74]	Uses a simple C++ parser and code generator (requiring Perl or Python) for test registration. * Has a framework for generating mocks of global functions, but not for generating mocks of objects.
Exercisix	No	No	No	No	No	Yes	Yes	Yes	Executables	[75]	BSD-licensed. Aimed to make adding tests as fast and easy as possible.
FCTX	Yes	Yes	Yes	No	No	No	Yes	No	Yes	[38]	Fast and complete unit testing framework all in one header. Declare and write your functions in one step. No dependencies. Cross platform.
Fructose	No	Yes	No	No	No	Yes	Yes	Yes	No	[76]	A simple unit test framework. See also Accu [77]
Name	**xUnit**	**Fixtures**	**Group fixtures**	**Generators**	**Mocks**	**Exceptions**	**Macros**	**Templates**	**Grouping**	**Homepage**	**Remarks**
Google C++ Mocking Framework					Yes	No	Yes	Yes		[78]	
Google C++ Testing Framework	Yes	Yes			Yes	Yes	Yes	Yes		[79]	Supports automatic test discovery, a rich set of assertions, user-defined assertions, death tests, fatal and non-fatal failures, various options for running the tests, and XML test report generation.

Hestia	No	Yes	Yes	No	No	Yes	Yes	No	Suites	[80]	Open source. Can test servers, libraries, and applications, and also test embedded software.
Igloo		Yes (Contexts)	No	No	No	Yes	Yes	Yes	Yes (nested contexts)	[81]	BDD style unit testing in C++
Isolator++											Commercial. Isolation/Mocking Framework for C++
mock++/mockcpp	Yes	Yes		No	Yes	Yes	Yes	Yes	Suites	[82]	Simple testing framework for C++ (requires cmake)
mockitopp					Yes					[83]	C++ implementation of mockito
mockpp	Yes	Yes		Yes	Yes	Yes	Yes	Yes	Suites	[84]	A C++ mocking framework hosted by Google
NanoCppUnit	No	Yes	Yes	No	No	No	Yes	Yes	Suites	[85]	Proof-of-concept
OAKUT	No	No	Yes	Yes (XML)	No	Yes	Yes	Yes	XML	[86]	Uses shared libraries / DLLs
QtTest	Yes	Yes	No	No	No	No	No	No		[87]	Built on the ultra cross platform Qt Library. Allows testing on Windows, MacOSX, Linux, BSD, Sybian, and any other platform Qt can build on.
QuickTest	No	No	No	No	No	Yes	Yes	Yes	No	[88]	
ShortCUT	No								Yes	[89]	
Symbian OS Unit	Yes									[90]	Based on CxxTest
TBrun										[91]	Commercial.
Tessy										[47]	Commercial.
TEST-DOG	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Suites	[92]	Open Source.
Test soon	No	Yes	Yes	Yes	No	Auto-detect	Yes	Yes	Macro (namespaces)	[93]	Complex macro logic (deliberately)
Testwell CTA++	Yes	No	No	Yes	No	Yes	Yes	Yes		[94]	Commercial.
tpunit++	Yes	Yes		No	No	Optional	Yes	Yes		[95]	A highly portable, simple C++ xUnit library contained in a single header.

Name	xUnit	Fixtures	Group fixtures	Generators	Mocks	Exceptions	Macros	Templates	Grouping	Homepage	Remarks
TUT	No	No	Yes	No	No	Yes	Yes	Yes	Templates	[96]	Based on templates. Automatic test registration/discovery, customizable reports generation process, various tests run options. Easy adaptable to work on Windows CE.
Unit++										[97]	
UnitTest	Yes	Yes	No	No	No	Yes	Yes	No	No	[98]	
UnitTest++	No	Yes	Yes		No	Yes	Yes	Yes	Suites	[99]	UnitTest++ is free software. Simplicity, portability, speed, and small footprint are all important aspects of UnitTest++.
UquoniTest	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	[100]	Commercial. Adds features like easy creation of (reusable) test directories, Rich Booleans in assertions, and more.
VectorCAST/C++										[101]	Commercial. Automated unit and integration testing, and code coverage.
Visual Assert	Yes	Yes	No	No	No	Yes	Yes	No		[102]	Unit-Testing Add-In for Visual Studio. Based on the cfix testing framework.
WinUnit										[103]	Focused more toward C/C++ than .NET developers
xTests						Yes	Yes			[56]	Depends on STLSoft C & C++ Libraries
Yaktest										[104]	

Cg

Name	xUnit	Homepage	Remarks
UnitTestCg	No	[105]	Pixel and Vertex Shaders supported.

CFML (ColdFusion)

Name	xUnit	Homepage	Remarks
CFUnit			
cfcUnit	Yes		
MXUnit	Yes		Unit Test Framework and Eclipse Plugin for CFML (ColdFusion)
MockBox	No		Mocking/Stubbing Framework for CFML (ColdFusion)

Clojure

Name	xUnit	Homepage	Remarks
Fact	Yes	[106]	

Cobol

Name	xUnit	Homepage	Remarks
COBOLUnit (CBU)	Yes	[107]	
savvytest	No	[108]	Commercial tool (rather than a framework) to edit and perform component tests on the IBM mainframe platform

Common Lisp

Name	xUnit	Homepage	Remarks
CLUnit		[109]	
CyberTiggyr Test		[110]	
FiveAM		[111]	
FReT		[112]	
grand-prix		[113]	
HEUTE		[114]	
LIFT		[115]	
lisp-unit		[116]	
RT		[117]	

stefil		[118]	
XLUnit		[119]	

Curl

Name	xUnit	Homepage	Remarks
CurlUnit		[120]	

Delphi

Name	xUnit	Homepage	Remarks
DUnit	Yes	[121]	-
DUnit2	Yes	[122]	-

Emacs Lisp

Name	xUnit	Homepage	Remarks
ElUnit		[123]	
elk-test	No	[124]	
Unit-test.el		[125]	

Erlang

Name	xUnit	Fixtures	Group Fixtures	Generators	Homepage	Remarks
EUnit	Yes	Yes	Yes	Yes	[126]	Included in Erlang release R12B5 and later

Fortran

Name	xUnit	Fixtures	Group fixtures	Generators	Homepage	Remarks
FUnit	Yes	Yes	Yes	No	[127]	Minimum Fortran plus assertions and boiler plate expanded via Ruby.
FRUIT	Yes	Yes	Yes	Yes	[128]	Written in Fortran 95 yet works with code written in any version of Fortran. Has assertions, fixture, setup, teardown, reporting and more. Generator in Ruby.
Ftnunit					[129]	
pFUnit					[130]	

ObjexxFTK:UnitTest					[131]	Easy: user only writes Fortran tests: Python script automatically finds all tests and generates the Fortran test driver. Supports F77-F2003. Assertion support and useful string and array libs included.
Lutin77					[132]	Running F77 unit tests. It is very light (97 lines of code) and only requires a C compiler. Hack it yourself and enjoy !

F#

Name	xUnit	Homepage	Remarks
FsCheck	No	[133]	Random testing (Fuzzing) combinator library based on QuickCheck for Haskell.
FsTest	Yes	[134]	Domain specific language for writing language oriented programming specifications in F#. Based on FsUnit syntax, but targeting xUnit.net.
NaturalSpec	No	[135]	Domain specific language for writing specifications in a natural language. Based on NUnit.
FsUnit	No	[136]	Stand-alone Behavior Driven Development framework, with natural syntax for writing specifications.
Other		-	See also listing for .NET programming languages, elsewhere on this page.

Groovy

Name	xUnit	Homepage	Remarks
easyb		[137]	BDD
geb		[138]	Functional testing framework
Spock	Yes	[139]	

Genexus

Name	xUnit	Homepage	Remarks
GXUnit	Yes	[140]	It's a framework (prototype) to perform unit testing for Genexus.

Haskell

Name	xUnit	Homepage	Remarks
HUnit	Yes	[141]	
QuickCheck		[142]	QuickCheck

HLSL

Name	xUnit	Homepage	Remarks
UnitTestCg	No	[105]	Pixel and Vertex Shaders supported.

ITT IDL

Name	xUnit	Homepage	Remarks
MGunit	Yes	[143]	
white paper		[144]	only a white paper, not a framework

Internet

Name	xUnit	Homepage	Remarks
HtmlUnit		[145]	Java headless browser emulator
HttpUnit			testing framework for web applications, typically used in combination with JUnit
IEUnit		[146]	testing framework for web applications, based on IE browser and Javascript
Canoo WebTest		[147]	
Selenium			Testing framework whose playback can run in most modern web browsers to test webpages.
soapUI		[148]	Open Source Web Service testing platform for Service Oriented Architectures.
SOAtest		[149]	Commercial. Testing platform whose record/playback runs in most modern web browsers to test webpages.

Java

Name	xUnit	Homepage	Remarks
JTiger			
SpryTest	Yes	[150]	Commercial. Automated Unit Testing Framework for Java
Jtest	Yes	[151]	Commercial. Automated software quality solution that includes unit test generation and execution as well as reporting industry standard code coverage.
JUnit	Yes	[152]	
JWalk			Fast, semi-automatic creation of exhaustive unit test-sets
TestNG	Yes		Actually an integration testing framework, which means its tests include unit tests, functional tests, and integration tests.

NUTester		[153]	Testing framework developed at Northeastern University to aid in teaching introductory computer science courses in Java
Concordion		[154]	Acceptance Test Driven Development
JExample	Yes	[155]	A JUnit extension that uses dependencies between test cases to reduce code duplication and improves defect localization.
DbUnit		[2]	A JUnit extension to perform unit testing with database-driven programs
JUnitEE		[156]	A JUnit extension for testing Java EE applications
Cactus			A JUnit extension for testing Java EE and web applications. Cactus tests are executed inside the Java EE/web container.
JSST		[157]	Java Server-Side Testing framework which is based on the similar idea to the one of Apache CACTUS, but unlike CACTUS it's not coupled to JUnit 3.x and can be used in conjunction with any testing framework.
GroboUtils		[158]	A JUnit extension providing Automated documentation, class hierarchy unit testing, code coverage, and multi-threaded tests.
Mockrunner		[159]	A JUnit extension for testing testing servlets, filters, tag classes and Struts actions and forms.
Unitils		[160]	Offers general utilities and features for helping with persistence layer testing and testing with mock objects. Offers specific support for testing application code that makes use of JPA, hibernate and spring. Unitils integrates with the test frameworks JUnit and TestNG.
JBehave		[161]	Behavior Driven Development
Instinct		[162]	Behavior Driven Development
JDave		[163]	Behavior Driven Development
beanSpec		[164]	Behavior Driven Development
XMLUnit		[1]	JUnit and NUnit testing for XML
EasyMock		[165]	A mock framework
JMock		[166]	A mock framework
Mockito		[167]	A mock framework, using the Hamcrest [168] library
Mockachino		[169]	A mock framework
Concutest		[170]	A framework for testing concurrent programs
SureAssert		[171]	An integrated Java unit testing solution for Eclipse. Contract-First Design and Test Driven Development

JavaScript

Name	xUnit	TAP	Client-side	Server-side	Homepage	Remarks
DOH			Yes	Yes	[172]	Dojo Objective Harness that can be run in-browser or independently via Rhino
JSUnit	Yes	No	Yes	No	[173]	JSUnit is no longer actively maintained
QUnit			Yes	Yes	[174]	jQuery test harness
Crosscheck			No	Yes	[175]	Browserless java-based framework
J3Unit			Yes	No	[176]	
JSNUnit			Yes	No	[177]	
YUI Test			Yes	No	[178]	
JSSpec			Yes	No	[179]	Behaviour-driven development framework
UnitTesting			Yes	No	[180]	script.aculo.us javascript test harness
JSpec			Yes	Yes	[181]	Highly readable BDD, 50+ matchers, DOM / framework independent, async, rhino, node.js support and more (no longer maintained)
Jasmine			Yes	Yes	[182]	BDD, framework independent, easy integration with Ruby projects and continuous builds. Allows for both DOM-less testing and asynchronous testing.
screw-unit			Yes	No	[183]	Requires jQuery
Test.Simple	No[184]	Yes	Yes	No	[18]	Write TAP-emitting unit tests in JavaScript and run them in your browser.
Test.More	No[1]	Yes	Yes	No	[185]	Write TAP-emitting unit tests in JavaScript and run them in your browser.
TestCase			Yes	No	[186]	
TestIt			Yes	Yes	[187]	Light-weight, non-polluting, and easy to setup and use
jsUnitTest			Yes	Yes[188]		Based on TestCase but without the Prototype dependency
JSTest			Yes	No	[189]	Light-weight, non-polluting browser-based framework
JSTest.NET			No	Yes	[190]	Browserless JavaScript unit test runner for use with MsTest, XUnit, NUnit, etc.
jsUnity	Yes	No	Yes	Yes	[191]	Context-agnostic (JavaScript, JScript (ASP/WSH), Rhino, etc.)
RhinoUnit			No	Yes	[192]	Rhino-based framework that allows tests to be run in Ant
JasUnit	Yes	No	Yes	No	[193]	Light-weight framework. Part of a project that provides Mocks and IoC.)
FireUnit			Yes	No	[194]	Testing framework that provides logging and viewing within a new tab of Firebug.
Js-test-driver	Yes		Yes	No	[195]	The goal of JsTestDriver is to build a JavaScript test runner which easily integrates with continuous builds systems and allows running tests on multiple browsers quickly to ease TDD style development.
Sinon.js	Compatible		Yes	Yes	[196]	Standalone test spies, stubs and mocks for JavaScript. No dependencies, works with any unit testing framework.

SOAtest	No		Yes	No	[149]	Commercial. Testing platform whose record/playback runs in most modern web browsers where client-side Javascript can be tested both through static analysis and functional verification.
Vows	No			Yes	[197]	
Nodeunit			Yes	Yes	[198]	Asynchronous Javascript testing framework

Lasso

Name	xUnit	Homepage	Remarks
L-Unit		[199]	

LaTeX

Name	xUnit	Homepage	Remarks
qsunit		[200]	

LabVIEW

Name	xUnit	Homepage	Remarks
LabVIEW Unit Test Framework	No		Generate test harnesses for VIs in LabVIEW automatically.
VI Tester	Yes	[201]	native LabVIEW object-oriented implementation of xUnit framework. Unit tests are written as VIs in LabVIEW.

LISP

Name	xUnit	Homepage	Remarks
FiveAM		[111]	
LIFT		[115]	
lisp-unit		[116]	

Lua

Name	xUnit	Homepage	Remarks
LuaUnit		[202]	
lunit		[203]	

MATLAB

Name	xUnit	Homepage	Remarks
mlUnit	Yes	[204]	
mlUnit_2008a	Yes	[205]	
Phelan's MUnit		[206]	The code is not available for download and an additional license restriction appears to violate the GPL, under which the code is purportedly licensed.
Lombardi's MUnit		[207]	Similar to xUnit
MATLAB xUnit Test Framework	Yes	[208]	MATLAB r2008a and later (uses OOP features introduced in this release). Renamed from **mtest**. Accepts both xUnit-style subclass or simple MATLAB function test cases.
Doctest	No	Bitbucket repository - source and documentation [209]	Allows automated test cases to be put in the documentation so your usage examples double as test cases and vice versa. A TAP producer. Inspired by the Python module of the same name. As of August 2011, it can only handle single line test-cases and its exception handling facility cannot handle exceptions that are generated after other output [210] .

MySQL

Name	xUnit	Homepage	Remarks
utMySQL	Yes	[211]	

.NET programming languages

Name	xUnit	Homepage	Remarks
csUnit	Yes		includes GUI, command line, VS2005 plug-in; supports C#, VB.NET, Managed C++, J#, other .NET languages, supports .NET 3.5 and earlier versions; integrated with ReSharper
DbUnit.NET		[212]	A .NET 2.0 unit testing framework for database access code
EMTF	No	[213]	open source
Gallio		[214]	Extensible, and neutral automation platform that provides a common object model, runtime services and tools (such as test runners) that may be leveraged by any number of test frameworks.
MbUnit	Yes	[215]	Extensible, model-based nUnit compatible framework. Part of the Gallio Test Automation Platform.
MSTest	No		A command-line tool for executing Visual Studio created unit tests outside of the Visual Studio IDE - not really a testing framework as it is a part of the Visual Studio Unit Testing Framework.
NaturalSpec	No	[135]	Domain specific language for writing specifications in a natural language. Based on NUnit.

NMate		[216]	NUnit and PartCover Code Generation and integration Addin for Microsoft Visual Studio 2005/2008
NUnit	Yes		includes GUI, command line, integrates into VisualStudio with ReSharper
NUnitAsp			Based on NUnit
Pex	Yes	[217]	Microsoft Research project providing White box testing for .NET, using the Z3 constraint solver to generate unit test input (rather than Fuzzing).
Quality Gate One Studio	No	[218]	Commercial/freeware test framework for unit and integration testing that analyses dependencies between test cases to flow data between them. Supports combinatorial testing, multithreading and time-dependencies.
QuickUnit.net	No	[219]	Implement unit tests without coding. Minimalist approach to test driven development.
Rhino Mocks	Yes	[220]	A dynamic mock object framework for the .NET platform.
Roaster	Yes	[221]	NUnit based framework and tools for the .NET Compact Framework
SpecFlow	Yes	[222]	Behavior Driven Development framework for .Net. Inspired by [[Cucumber (software)|Cucumber]. Inegrates with NUnit, MSTest, MbUnit, and others.
Specter	Yes	[223]	Behavior Driven Development with an easy and readable syntax for writing specifications. Includes command line, optional integration with NUnit
TestDriven.NET		[224]	[commercial]
.TEST	Yes	[225]	Commercial. Automated software quality solution that includes unit test generation and execution as well as reporting industry standard code coverage.
Typemock Isolator	Yes	[226]	Commercial unit testing framework with simple API and test code generation features, supports C#, ASP.NET, SharePoint, Silverlight.
Visual Studio	No		The Visual Studio Unit Testing Framework is included with Visual Studio Team System 2005 and later editions, integrated with Visual Studio IDE. It is not included with Visual Studio Standard Edition or Visual Studio Express editions.
Visual T#	Yes	[227]	Visual T# is a unit testing framework and development environment integrated with Visual Studio. It includes T#, a programming language designed specifically to naturally express unit test intentions, and tools for compiling, running and maintaining them.
xUnit.net	Yes	[228]	Developed by the original inventor of NUnit to be its successor. xUnit.net is currently the highest rated .NET unit testing framework [229] due to it being leaner with a more refined syntax and lower friction usage then NUnit.

Objective-C

Name	xUnit	Homepage	Remarks
GHUnit		[230]	A easy to integrate, use and visual test framework for simulator and devices.
CATCH		[231]	A modern, fully featured, unit test framework with no external dependencies - all implemented in headers
Cedar		[232]	BDD for Objective-C
iPhone Unit Testing		[233]	Unit testing framework based on OCUnit that works on the iPhone simulator and device.
Kiwi		[234]	RSpec-style BDD for Objective-C with support for mocks and stubs.
ObjcUnit		[235]	
OCUnit		[236]	

WiteBox for iPhone		[237]	Exclusively for iPhone unit testing, provides visual feedback, hierarchical results display, and test durations.
WOTest		[238]	

Ocaml

Name	xUnit	TAP	Homepage	Remarks
OUnit			[239]	Based on HUnit, which is based on JUnit
TestSimple		Yes	[240]	Generates TAP
FORT			[241]	

Object Pascal (Free Pascal)

Name	xUnit	TAP	Homepage	Remarks
FPCUnit	Yes	No	[242]	This is a port to Free Pascal of the JUnit core framework.
Tap4Pascal	No	Yes	[243]	A Pascal implementation of the Test Anything Protocol
FPTest	Yes	No	[244]	This is a fork of DUnit2, specifically for use with the Free Pascal Compiler.

PegaRULES Process Commander

Name	xUnit	Homepage	Remarks
PRUnit	Yes	[http://prunit.sourceforge.net/	xUnit style testing adapted to PRPC

Perl

Name	xUnit	TAP	Homepage	Remarks
TAP	N/A	Yes	[245]	the Test Anything Protocol used by most Perl tests
Test::Harness	N/A	Yes	[246]	the standard Perl test runner and TAP parser
Test::More	No	Yes	[247]	The baseline testing module, included with all modern Perl installations
Test::Class	Yes	Yes	[248]	xUnit style testing adapted to Perl
Test::Builder	N/A	Yes	[249]	a module for making more testing modules. These modules can be combined in a single test program
Test::Unit (a.k.a. PerlUnit)	Yes	No	[250] ([251])	a fairly straight port of JUnit to Perl. **Note:** This code seems to be abandoned as noted here [252] and here [253].
Test::DBUnit	N/A	Yes	[254]	The database testing modules for both clear box and black box testing
Test::Unit::Lite	Yes	Yes (via Test::Unit::HarnessUnit)	[255]	Test::Unit replacement without external dependencies and with some bugs fixed
Test::Able	Yes	Yes	[256]	xUnit style testing with Moose [257]

PHP

Name	xUnit	TAP	Homepage	Remarks
PHPUnit	Yes	Yes	[13]	
PHP Unit Testing Framework	Yes	No	[258]	It produces ASCII, XML or XHTML Output and runs from the command line.
SimpleTest	Yes	No	[259]	
Testilence	Yes	No	[260]	
lime	No	Yes	[261]	Sub-project of symfony
Apache-Test	No	Yes	[14]	PHP implementation of Test::More (test-more.php) Info [262]
ojes	No	No	[263]	Documentation Driven Testing
SnapTest	Yes	Yes	[24]	
OnionTest	No	Yes	[264]	Write an Onion! No coding needed just some txt files.

PL/SQL

Name	xUnit	Homepage	Remarks
utPLSQL	Yes	utSQL [265]	Initially developed by Steven Feuerstein, an open source unit testing framework for PL/SQL development modeled on the Junit and xUnit frameworks.
Quest Code Tester for Oracle		[266]	A commercial PL/SQL testing tool from Quest Software (the makers of Toad) and Steven Feuerstein.
pl/unit		[267]	
PL/SQL Unit Testing for Oracle (PLUTO)	Yes	[22]	Open source unit testing framework modeled after JUnit and based on the Oracle PL/SQL object system
ruby-plsql-spec		[268]	PL/SQL unit testing with Ruby open source libraries
DBFit		[269]	DbFit is a set of FIT fixtures which enables FIT/FitNesse tests to execute directly against a database.

PostgreSQL

Name	xUnit	TAP	Homepage	Remarks
Epic			[270]	Epic is a unit test framework for PostgreSQL stored procedures. It requires PG 8.1 or higher.
pgTAP	Yes	Yes	[271]	Write tests in SQL or xUnit-style functions.
PGtools			[272]	Schema loading and testing tools for PostgreSQL.
PGUnit	Yes	No	[273]	xUnit-style framework for stored procedures.

PowerBuilder

Name	xUnit	Homepage	Remarks
PBUnit		[274]	

Progress 4GL

Name	xUnit	Homepage	Remarks
proUnit	Yes	[275]	xUnit-style framework for Progress OpenEdge procedures and objects

Prolog

Name	xUnit	Homepage	Remarks
PlUnit		[276]	

Python

Name	xUnit	Generators	Fixtures	Group Fixtures	Homepage	Remarks
PyUnit	Yes	Yes	Yes	No		it's part of Python's standard library
XPyUnit						adding XML report generation to PyUnit
TestOOB						an extended test framework for PyUnit
Doctest						easy, Pythonic, and part of Python's standard library
Nose	Yes	Yes	Yes		[277]	a discovery-based unittest extension
py.test	Yes	Yes	Yes	Yes	[278]	distributed testing tool
TwistedTrial	Yes	Yes	Yes	No	[279]	PyUnit extensions for asynchronous and event-driven code

R programming language

Name	xUnit	Homepage	Remarks
RUnit	No	[280]	Open source
testthat	No	[281]	Open source

REALbasic

Name	xUnit	Homepage	Remarks
RBUnit	No	[282]	

Rebol

Name	xUnit	Homepage	Remarks
Runit		[283]	

RPG

Name	xUnit	Homepage	Remarks
RPGUnit	Yes	[284]	

Ruby

Name	xUnit	Homepage	Remarks
Test::Unit	Yes	[285]	
RSpec		Behaviour-driven development framework	
Shoulda		[286]	
microtest			
Bacon			
minitest		[287] [288]	Ruby Gem by Ryan Davis

SAS

Name	xUnit	Homepage	Remarks
FUTS	Yes	[289]	the Framework for Unit Testing SAS
SCLUnit	Yes	[290]	SAS/AF implementation of the xUnit unit testing framework SAS

Scala

Name	xUnit	Homepage	Remarks
Rehersal (sic)		[291]	with JMock like Expectations and natural language test names
ScUnit		[292]	JUnit style testing with fixture method injection and matchers for assertion
specs		[293]	Behavior Driven Development
ScalaCheck		[294]	Similar to QuickCheck

Scheme

Name	xUnit	Homepage	Remarks
SchemeUnit	Yes	[295]	

Shell

Name	xUnit	Homepage	Remarks
assert.sh		[296]	LGPL licensed. Lightweight..
ATF		[22]	BSD Licensed. Originally developed for the NetBSD operating system but works well in most Unix-like platforms. Ability to install tests as part of a release.
Roundup	No	[297]	
ShUnit	Yes	[298]	
shUnit2	Yes	[299]	Originally developed for log4sh
filterunit		[300]	Test framework for filters and other command-line programs
Tap-functions		[301]	A TAP-producing BASH library

Simulink

Name	xUnit	Homepage	Remarks
slUnit		[302]	
TPT	Yes	[50]	Time Partition Testing: Automated interface analysis, testframe generation, test execution, test assessment, reporting.

Smalltalk

Name	xUnit	Homepage	Remarks
SUnit	Yes		The original source of the xUnit design

SQL

Name	xUnit	Homepage	Remarks
SQLUnit		[303]	
DbFit		[269]	Compare FitNesse.

TargetLink

Name	xUnit	Homepage	Remarks
TPT		[50]	Time Partition Testing: Automated interface analysis, testframe generation, test execution, test assessment, reporting.

Tcl

Name	xUnit	Homepage	Remarks
tcltest		[304]	
tclUnit	Yes	[305]	

TinyOS/nesC

Name	xUnit	Homepage	Remarks
TUnit	Yes	[306]	Embedded multi-platform wireless testing and characterization

Transact-SQL

Name	xUnit	Homepage	Remarks
TSQLUnit	Yes	[307]	
utTSQL	Yes	[308]	
Visual Studio Team Edition for Database Professionals		[309]	
T.S.T.		[310]	
Slacker [311]	Yes	[311]	Based on RSpec

Visual FoxPro

Name	xUnit	Homepage	Remarks
FoxUnit		[312]	

Visual Basic (VB6)

For unit testing frameworks for VB.NET, see the .NET programming languages section.

Name	xUnit	Homepage	Remarks
vbUnit		[313]	Visual Basic and COM objects
vbUnitFree		[314]	Visual Basic and COM objects
VbaUnit		[315]	Visual Basic for Applications
ExcelVbaUnit		[316]	Similar to VbaUnit, but specifically for testing Excel VBA (written as an Excel add-in)
TinyUnit		[317]	Visual Basic 6, VB .NET, and PHP5
SimplyVBUnit	Yes	[318]	VB6 Unit Testing Framework modeled after the popular NUnit for .NET

Visual Lisp

Name	xUnit	Homepage	Remarks
vl-unit		[319]	Unit testing framework for Visual Lisp.

XML

Name	xUnit	Homepage	Remarks
XUnit		[320]	for testing • native XML programs, • individual XSLT templates, • and Java programs that deal with XML data
WUnit		[321]	for testing Web applications • tests are written in XML/XPath (XUnit), • AJAX applications are supported, • can also test server-side functionalities if they are made with Java servlets (for example, it is possible to store authoritatively an object in the user session server-side without sending an HTTP request and then get with HTTP the page that renders it)
SOAtest	No	[149]	Commercial. Parasoft's full-lifecycle quality platform for ensuring secure, reliable, compliant business processes.
Vibz Automation	No	[322]	Open Source. Vibzworld's Fully featured Open source test framework.

Name	xUnit	Homepage	Remarks
AntUnit		[323]	for testing Apache Ant tasks

XSLT

Name	xUnit	Homepage	Remarks
juxy		[324]	a library for unit testing XSLT stylesheets from Java
Tennison Tests		[325]	allows to write unit-tests in XML, exercising XSLT from Apache Ant. When incorporated in a continuous integration environment, it allows to run multiple XSLT tests as part of a build, failing the build when they go wrong.
UTF-X		[326]	unit testing framework for XSLT that strongly supports the test-first-design principle with test rendition and test validation features
XMLUnit	Yes	[1]	Plugin for JUnit and NUnit, allowing Assertion-style tests to be written for XSLT documents in Java or C#
XSLTunit		[327]	proof of concept unit testing framework for XSLT

Other

Name	xUnit	Homepage	Remarks
Test Manager		[328]	
IdMUnit	Yes	[329]	Identity management

References

[1] http://help.sap.com/saphelp_nw2004s/helpdata/en/a2/8a1b602e858645b8aac1559b638ea4/frameset.htm
[2] http://code.google.com/p/as3flexunitlib/
[3] http://www.flexunit.org
[4] http://code.google.com/p/reflex-unit/
[5] http://www.funit.org/
[6] http://code.google.com/p/astuce/
[7] http://www.asunit.org/
[8] http://code.google.com/p/dpuint/
[9] http://fluint.googlecode.com/
[10] http://code.google.com/p/mojotest/
[11] http://libre.adacore.com/libre/tools/aunit/
[12] http://www.ipl.com/adatest
[13] http://ahven.stronglytyped.org/
[14] http://www.ldra.com/tbrun.asp
[15] http://www.vectorcast.com
[16] http://nirs.freeshell.org/asunit/
[17] http://applemods.sourceforge.net/mods/Development/ASTest.php
[18] http://aspunit.sourceforge.net/
[19] http://portal.acm.org/citation.cfm?id=1145723t
[20] http://aceunit.sourceforge.net/
[21] http://ispras.linux-foundation.org/index.php/API_Sanity_Autotest
[22] http://www.NetBSD.org/~jmmv/atf/
[23] http://autounit.tigris.org/
[24] http://www.parasoft.com/jsp/solutions/cpp_solution.jsp?itemId=340
[25] http://www.ipl.com/products/tools/pt413.php
[26] http://www.agilerules.com/projects/catsrunner/index.phtml
[27] http://www.cfix-testing.org
[28] http://www.lastcraft.com/cgreen.php
[29] http://check.sourceforge.net/
[30] http://code.google.com/p/cmockery/
[31] http://cu.danfis.cz/
[32] http://cunit.sourceforge.net/
[33] http://code.google.com/p/cunitwin32/
[34] http://www.falvotech.com/content/cut/
[35] http://cutest.sourceforge.net/
[36] http://cutter.sourceforge.net/
[37] http://embunit.sourceforge.net/
[38] http://fctx.wildbearsoftware.com
[39] http://library.gnome.org/devel/glib/2.20/glib-Testing.html
[40] https://garage.maemo.org/projects/gunit
[41] http://koanlogic.com/libu
[42] http://www.jera.com/techinfo/jtns/jtn002.html
[43] http://rcunit.sourceforge.net
[44] http://www.rational.com
[45] http://seatest.googlecode.com
[46] http://www.accord-soft.com/dynamicanalyser.html
[47] http://www.hitex.de/perm/tessy.htm
[48] http://www.testape.com

[49] http://test-dept.googlecode.com
[50] http://www.piketec.com/products/tpt.php?lang=en
[51] http://unity.sourceforge.net
[52] http://cmock.sourceforge.net
[53] http://cexception.sourceforge.net
[54] http://ceedling.sourceforge.net
[55] http://www.visualassert.com
[56] http://xtests.sourceforge.net/
[57] http://code.google.com/p/lcut/
[58] http://aeryn.tigris.org/
[59] Llopis, Noel. "Exploring the C++ Unit Testing Framework Jungle" (http://gamesfromwithin.com/exploring-the-c-unit-testing-framework-jungle#boost), 2004-12-28. Retrieved on 2010-2-13.
[60] Rozental, Gennadiy "Boost Test Fixture Documentation" (http://www.boost.org/doc/libs/1_42_0/libs/test/doc/html/utf/user-guide/fixture.html). Retrieved on 2010-2-13.
[61] Rozental, Gennadiy "Boost Test Test Suite Level Fixture Documentation" (http://www.boost.org/doc/libs/1_42_0/libs/test/doc/html/utf/user-guide/fixture/test-suite-shared.html). Retrieved on 2010-2-13.
[62] http://www.boost.org/doc/libs/1_42_0/libs/test/doc/html/index.html
[63] http://www.ipl.com/products/tools/pt411.php
[64] https://github.com/philsquared/Catch
[65] http://www.cfix-testing.org/
[66] http://cput.codeplex.com/
[67] http://cpptest.sourceforge.net/
[68] http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page
[69] http://sourceforge.net/projects/cpputest
[70] http://c2.com/cgi/wiki?CppUnitLite
[71] http://cpunit.sourceforge.net/
[72] http://www.cute-test.com/
[73] http://codesink.org/cutee_unit_testing.html
[74] http://cxxtest.sourceforge.net/
[75] http://alexanderchuranov.com/software/exercisix/
[76] http://sourceforge.net/projects/fructose/
[77] http://accu.org/index.php/journals/1305
[78] http://code.google.com/p/googlemock/
[79] http://code.google.com/p/googletest/
[80] http://sourceforge.net/projects/hestia/
[81] http://igloo-testing.org
[82] http://code.google.com/p/mockcpp/
[83] http://code.google.com/p/mockitopp/
[84] http://mockpp.sourceforge.net/
[85] http://www.xpsd.org/cgi-bin/wiki?NanoCppUnit
[86] http://www.oaklib.org/oakut/index.html
[87] http://doc.qt.nokia.com/latest/qtestlib-tutorial.html
[88] http://quicktest.sf.net
[89] http://www.codeproject.com/KB/applications/shortcut.aspx?print=true
[90] http://www.symbianosunit.co.uk/
[91] http://www.ldra.co.uk/tbrun.asp
[92] http://www.bigangrydog.com/testdog/unit_testing.xhtml
[93] http://testsoon.sourceforge.net
[94] http://www.testwell.fi/ctadesc.html
[95] http://github.com/tpounds/tpunitpp
[96] http://tut-framework.sourceforge.net/
[97] http://unitpp.sourceforge.net/
[98] http://devmentor.org#UnitTest
[99] http://unittest-cpp.sourceforge.net/
[100] http://www.q-mentum.com/uquonitest.php
[101] http://www.vectorcast.com/
[102] http://www.visualassert.com/
[103] http://winunit.codeplex.com/
[104] http://yaktest.sourceforge.net/

[105] http://code.google.com/p/unittestcg/source
[106] http://github.com/weavejester/fact/tree/
[107] http://sites.google.com/site/cobolunit/
[108] http://www.savignano.net/savvytest
[109] http://www.ancar.org/CLUnit/docs/CLUnit.html
[110] http://cybertiggyr.com/gene/lut/
[111] http://common-lisp.net/project/bese/FiveAM.html
[112] http://common-lisp.net/project/fret/
[113] http://common-lisp.net/project/grand-prix/
[114] http://www.rdrop.com/~jimka/lisp/heute/heute.html
[115] http://common-lisp.net/project/lift/
[116] http://www.cs.northwestern.edu/academics/courses/325/readings/lisp-unit.html
[117] http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/testing/rt/0.html
[118] http://common-lisp.net/project/stefil/
[119] http://www.cliki.net/xlunit
[120] http://www.curl.com
[121] http://dunit.sourceforge.net/
[122] http://sourceforge.net/projects/dunit2/
[123] http://www.emacswiki.org/cgi-bin/wiki/ElUnit
[124] http://www.emacswiki.org/cgi-bin/wiki/ElkTest
[125] http://www.emacswiki.org/cgi-bin/wiki/unit-test.el
[126] http://svn.process-one.net/contribs/trunk/eunit/doc/overview-summary.html
[127] http://nasarb.rubyforge.org/funit
[128] http://sourceforge.net/projects/fortranxunit
[129] http://flibs.sf.net/ftnunit.html
[130] http://sourceforge.net/projects/pfunit/
[131] http://objexx.com/ObjexxFTK.html
[132] http://www.logilab.org/project/lutin77
[133] http://www.codeplex.com/fscheck/
[134] http://www.codeplex.com/FsTest
[135] http://bitbucket.org/forki/naturalspec/wiki/Home
[136] http://fsunit.codeplex.com/
[137] http://www.easyb.org/
[138] http://geb.codehaus.org/
[139] http://code.google.com/p/spock/
[140] http://wiki.gxtechnical.com/commwiki/servlet/hwiki?GXUnit
[141] http://hunit.sourceforge.net
[142] http://www.cs.chalmers.se/~rjmh/QuickCheck/
[143] http://mgunit.idldev.com
[144] http://www.barnett.id.au/idl/UnitRun.html
[145] http://htmlunit.sourceforge.net/
[146] http://code.google.com/p/ieunit/
[147] http://webtest.canoo.com/
[148] http://www.soapui.org/
[149] http://www.parasoft.com/jsp/solutions/soa_solution.jsp
[150] http://www.sprystone.com
[151] http://www.parasoft.com/jsp/products/jtest.jsp?itemId=14
[152] http://junit.org/
[153] http://www.ccs.neu.edu/javalib
[154] http://www.concordion.org
[155] http://scg.unibe.ch/research/jexample/
[156] http://www.junitee.org/
[157] https://github.com/shyiko/jsst/wiki
[158] http://groboutils.sourceforge.net/
[159] http://mockrunner.sourceforge.net/
[160] http://www.unitils.org
[161] http://jbehave.org/
[162] http://code.google.com/p/instinct/
[163] http://www.jdave.org/

[164] http://sourceforge.net/projects/beanspec
[165] http://easymock.org/
[166] http://www.jmock.org/
[167] http://code.google.com/p/mockito/
[168] http://code.google.com/p/hamcrest/
[169] http://code.google.com/p/mockachino/
[170] http://www.concutest.org/
[171] http://www.sureassert.com/
[172] http://www.dojotoolkit.org/reference-guide/util/doh.html
[173] http://www.jsunit.net/
[174] http://docs.jquery.com/QUnit
[175] http://thefrontside.net/crosscheck
[176] http://j3unit.sourceforge.net/
[177] http://www.valleyhighlands.com/testingframeworks/
[178] http://developer.yahoo.com/yui/yuitest/
[179] http://jania.pe.kr/aw/moin.cgi/JSSpec
[180] http://github.com/madrobby/scriptaculous/wikis/unit-testing
[181] http://visionmedia.github.com/jspec
[182] http://pivotal.github.com/jasmine
[183] http://github.com/nkallen/screw-unit/tree/master
[184] TAP output can easily be transformed into JUnit XML via the CPAN module TAP::Formatter::JUnit.
[185] http://openjsan.org/doc/t/th/theory/Test/Simple/0.21/lib/Test/More.html
[186] http://rubyforge.org/projects/testcase/
[187] http://github.com/DouglasMeyer/test_it
[188] TAP available with the latest repository version http://code.google.com/p/jsunity/source/browse/trunk/jsunity/jsunity.js
[189] http://github.com/willurd/JSTest
[190] http://jstest.codeplex.com
[191] http://jsunity.com/
[192] http://code.google.com/p/rhinounit/
[193] http://code.google.com/p/jasproject/
[194] http://fireunit.org/
[195] http://code.google.com/p/js-test-driver/
[196] http://cjohansen.no/sinon/
[197] http://vowsjs.org
[198] https://github.com/caolan/nodeunit
[199] http://www.l-unit.org/
[200] http://www.ctan.org/tex-archive/help/Catalogue/entries/qstest.html
[201] http://jkisoft.com/vi-tester
[202] http://phil.freehackers.org/programs/luaunit/index.html
[203] http://www.nessie.de/mroth/lunit/
[204] http://mlunit.sourceforge.net
[205] http://www.mathworks.com/matlabcentral/fileexchange/21888
[206] http://xtargets.com/cms/Tutorials/Matlab-Programming/MUnit-Matlab-Unit-Testing.html
[207] http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=11306&objectType=File
[208] http://www.mathworks.com/matlabcentral/fileexchange/22846-matlab-xunit-test-framework
[209] https://bitbucket.org/tgs/doctest-for-matlab/overview
[210] Smith, Thomas. "Doctest - embed testable examples in your function's help comments" (https://bitbucket.org/tgs/doctest-for-matlab/overview). . Retrieved 5 August 2011.
[211] http://utmysql.sourceforge.net/
[212] http://dbunit-net.sourceforge.net/
[213] http://emtf.codeplex.com/
[214] http://gallio.org
[215] http://mbunit.com
[216] http://code.google.com/p/nmate
[217] http://research.microsoft.com/en-us/projects/Pex
[218] http://www.qgonestudio.com
[219] http://www.quickunit.com/
[220] http://www.ayende.com/projects/rhino-mocks.aspx
[221] http://www.codeplex.com/roaster

[222] http://specflow.org/
[223] http://specter.sf.net
[224] http://www.testdriven.net
[225] http://www.parasoft.com/jsp/solutions/dotnet_solution.jsp?itemId=342
[226] http://www.typemock.org
[227] http://www.prettyobjects.com/en/Products/TSharp
[228] http://xunit.codeplex.com
[229] http://stackoverflow.com/questions/261139/nunit-vs-mbunit-vs-mstest-vs-xunit-net
[230] https://github.com/gabriel/gh-unit
[231] https://github.com/philsquared/Catch/wiki/Catch-for-Objective-C
[232] https://github.com/pivotal/cedar
[233] http://code.google.com/p/google-toolbox-for-mac/wiki/iPhoneUnitTesting
[234] http://kiwi-lib.info/
[235] http://oops.se/objcunit/
[236] http://www.sente.ch/software/ocunit/
[237] http://code.google.com/p/witebox/
[238] http://test.wincent.com/
[239] http://www.xs4all.nl/~mmzeeman/ocaml
[240] http://www.iinteractive.com/ocaml/
[241] http://sourceforge.net/projects/fort
[242] http://camelos.sourceforge.net/fpcUnit.html
[243] http://sourceforge.net/projects/tap4pascal
[244] http://github.com/graemeg/fptest
[245] http://search.cpan.org/perldoc/TAP
[246] http://search.cpan.org/perldoc/Test::Harness
[247] http://search.cpan.org/perldoc/Test::More
[248] http://search.cpan.org/perldoc/Test::Class
[249] http://search.cpan.org/perldoc/Test::Builder
[250] http://search.cpan.org/perldoc/Test::Unit
[251] http://perlunit.sourceforge.net/
[252] http://www.nntp.perl.org/group/perl.qa/2005/02/msg3562.html
[253] http://www.nntp.perl.org/group/perl.qa/2005/02/msg3571.html
[254] http://search.cpan.org/perldoc/Test::DBUnit
[255] http://search.cpan.org/perldoc/Test::Unit::Lite
[256] http://search.cpan.org/perldoc/Test::Able
[257] http://search.cpan.org/perldoc/Moose
[258] http://php-unit-test.sourceforge.net/
[259] http://www.simpletest.org/
[260] http://www.testilence.org/
[261] http://trac.symfony-project.com/trac/browser/tools/lime/trunk
[262] http://shiflett.org/blog/2006/jan/test-simple-for-php
[263] http://ojesunit.blogspot.com/
[264] http://www.oniontest.org/
[265] http://utplsql.sourceforge.net/
[266] http://unittest.inside.quest.com/index.jspa
[267] http://plunit.com/
[268] http://github.com/rsim/ruby-plsql-spec
[269] http://www.fitnesse.info/dbfit
[270] http://www.epictest.org/
[271] http://pgtap.org/
[272] http://sourceforge.net/projects/pgtools/
[273] http://en.dklab.ru/lib/dklab_pgunit/
[274] http://web.archive.org/web/20090728170628/http://geocities.com/pbunit/
[275] http://prounit.sourceforge.net
[276] http://www.swi-prolog.org/packages/plunit.html
[277] http://somethingaboutorange.com/mrl/projects/nose/
[278] http://pytest.org
[279] http://twistedmatrix.com/trac/wiki/TwistedTrial
[280] http://sourceforge.net/projects/runit/

[281] http://cran.r-project.org/web/packages/testthat/index.html
[282] http://logicalvue.com/blog/2007/02/rbunit-is-now-free/
[283] http://www.rebol.org
[284] http://rpgunit.sourceforge.net/
[285] http://www.ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html
[286] http://www.thoughtbot.com/projects/shoulda
[287] http://rubydoc.info/gems/minitest/2.0.2/frames
[288] http://blog.zenspider.com/minitest/
[289] http://thotwave.com/products/futs.jsp
[290] http://www.sascommunity.org/mwiki/index.php?title=SclUnit&redirect=no
[291] http://rehersal.sourceforge.net
[292] http://code.google.com/p/scunit/
[293] http://code.google.com/p/specs/
[294] http://code.google.com/p/scalacheck/
[295] http://planet.plt-scheme.org/display.ss?package=schemeunit.plt&owner=schematics
[296] https://github.com/lehmannro/assert.sh
[297] http://itsbonus.heroku.com/p/2010-11-01-roundup
[298] http://shunit.sourceforge.net
[299] http://code.google.com/p/shunit2/
[300] http://www.merten-home.de/FreeSoftware/filterunit/
[301] http://testanything.org/wiki/index.php/Tap-functions
[302] http://mlunit.sourceforge.net/index.php/The_slUnit_Testing_Framework
[303] http://sqlunit.sourceforge.net/
[304] http://www.tcl.tk/man/tcl8.4/TclCmd/tcltest.htm
[305] http://sourceforge.net/projects/tclunit/
[306] http://www.lavalampmotemasters.com
[307] http://tsqlunit.sourceforge.net/
[308] http://utTSQL.sourceforge.net/
[309] http://www.microsoft.com/downloads/details.aspx?FamilyID=7DE00386-893D-4142-A778-992B69D482AD&displaylang=en
[310] http://TST.Codeplex.com/
[311] http://github.com/vassilvk/slacker/wiki/
[312] http://www.foxunit.org/
[313] http://vbunit.com/
[314] http://vbunitfree.sourceforge.net/
[315] http://www.c2.com/cgi/wiki?VbaUnit
[316] http://code.google.com/p/excelvbaunit/
[317] http://www.w-p.dds.nl/tinyunit.htm
[318] http://simplyvbunit.sourceforge.net/
[319] http://code.google.com/p/vl-unit/
[320] http://reflex.gforge.inria.fr/xunit.html
[321] http://reflex.gforge.inria.fr/wunit.html
[322] http://code.google.com/p/vauto/
[323] http://ant.apache.org/antlibs/antunit/
[324] http://juxy.tigris.org/
[325] http://tennison-tests.sourceforge.net/
[326] http://utf-x.sourceforge.net/
[327] http://xsltunit.org/
[328] http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=366928&fromcon
[329] http://idmunit.sourceforge.net

External links

- Oracle Unit Testing - tutorial site (http://www.oracleunittesting.com/)
- Other list of various unit testing frameworks (http://xprogramming.com/software)
- OpenSourceTesting.org lists many unit testing frameworks, performance testing tools and other tools programmers/developers may find useful (http://opensourcetesting.org/)
- Testing Framework (http://c2.com/cgi/wiki?TestingFramework)

SUnit

SUnit is a unit testing framework for the programming language Smalltalk. It is the original source of the xUnit design, originally written by the creator of Extreme Programming, Kent Beck. SUnit allows writing tests and checking results in Smalltalk. The resulting tests are very stable, but this method has the disadvantage that testers must be able to write simple Smalltalk programs.

History

Originally published as chapter 30 "Simple Smalltalk Testing", in the book **Kent Beck's Guide to Better Smalltalk** by Kent Beck, Donald G. Firesmith (Editor) (Publisher: Cambridge University Press, Pub. Date: December 1998, ISBN 9780521644372, 408pp)

External links

- Official website [1] @ Camp Smalltalk
- Sunit @ Ward Cunningham's Wiki [2]
- Kent Beck's original SUnit paper [3]

References

[1] http://http://sunit.sourceforge.net
[2] http://c2.com/cgi/wiki?SmalltalkUnit

JUnit

JUnit

Developer(s)	Kent Beck, Erich Gamma, David Saff
Stable release	4.8.2 / April 8, 2010
Preview release	4.9 Beta 2 / January 21, 2011
Written in	Java
Operating system	Cross-platform
Type	Unit testing tool
License	Common Public License
Website	http://junit.sourceforge.net

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit that originated with SUnit.

JUnit is linked as a JAR at compile-time; the framework resides under packages `junit.framework` for JUnit 3.8 and earlier and under `org.junit` for JUnit 4 and later.

Example of JUnit test fixture

A JUnit Test fixture inherits from `junit.framework.TestCase`. Test methods must be annotated by the `@Test` annotation. It is also possible to define a method to execute before (or after) each (or all) of the test methods with the `@Before` (or `@After`) and `@BeforeClass` (or `@AfterClass`) annotations.[1]

```java
import junit.framework.TestCase;
import org.junit.*;

public class TestFoobar extends TestCase{
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }

    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }

    @After
    public void tearDown() throws Exception {
```

```
        // Code executed after each test
    }

    @Test
    public void test() {
        assertTrue(true);
    }
}
```

Ports

JUnit has been ported to other languages including:

- Actionscript (FlexUnit [2])
- Ada (AUnit [11])
- C (CUnit [32])
- C# (NUnit)
- C++ (CPPUnit)
- Fortran (fUnit)
- Delphi (DUnit)
- Free Pascal (FPCUnit [242])
- JavaScript (JSUnit)
- Objective-C (OCUnit [236])
- Perl (Test::Class [3] and Test::Unit [4])
- PHP (PHPUnit)
- Python (PyUnit)
- R (RUnit [5])
- Haskell (HUnit [6])
- Qt (QTestLib)

References

[1] Kent Beck, Erich Gamma. "JUnit Cookbook" (http://junit.sourceforge.net/doc/cookbook/cookbook.htm). junit.sourceforge.net. . Retrieved 2011-05-21.
[2] http://www.flexunit.org/
[3] http://search.cpan.org/~adie/Test-Class-0.31/lib/Test/Class.pm
[4] http://search.cpan.org/~mcast/Test-Unit-0.25/lib/Test/Unit.pm
[5] http://RUnit.sourceforge.net/
[6] http://hackage.haskell.org/package/HUnit

External links

- JUnit home page (http://junit.org/)
- JUnit antipatterns (developerWorks) (http://www.ibm.com/developerworks/opensource/library/os-junit/?ca=dgr-lnxw07JUnite) and JUnit antipatterns (Exubero) (http://www.exubero.com/junit/antipatterns.html)
- An early look at JUnit 4 (http://www.ibm.com/developerworks/java/library/j-junit4.html?ca=dgr-lnxw01JUnit4)
- JUnit Presentation (http://www.methodsandtools.com/tools/tools.php?junit)
- JUnits (http://www.bunker37.com/2011/02/junits/)

CppUnit

CppUnit

Stable release	1.12.1 / February 19, 2008
Written in	C++
Type	Unit testing tool
License	GNU Lesser General Public License
Website	https://launchpad.net/cppunit2

CppUnit is a unit testing framework module for the C++ programming language, described as a C++ port of JUnit. The library is released under the GNU Lesser General Public License. The library can be compiled for a variety of POSIX platforms, allowing unit-testing of 'C' sources as well as C++ with minimal source modification. The framework has a neutral UI, running tests in suites. Test result output is sent to a filter, the most basic being a simple pass or fail count printed out, or more advanced filters allowing XML output compatible with continuous integration reporting systems .

External links

- Project's site [1].
- Llopis, Noel (2004-12-28). "Exploring the C++ Unit Testing Framework Jungle" [2].
- Unit-tests with C++ using the framework CppUnit [3]
- MiniCppUnit [4]: Another C++ port with a minimalistic approach.
- Unit++ [5]: A Unit-Testing framework designed to be an alternative to CppUnit, because of its focus in being more adapted to C++ language, instead of being a port of JUnit to C++.

Books

Game Programming Gems 6 (ISBN 1-58450-450-1) contains an article called "Using CPPUnit to implement unit testing" by Blake Madden

References

[1] https://launchpad.net/cppunit2
[2] http://www.gamesfromwithin.com/articles/0412/000061.html
[3] http://www.evocomp.de/tutorials/tutorium_cppunit/howto_tutorial_cppunit_en.html
[4] http://www.dtic.upf.edu/~parumi/MiniCppUnit/
[5] http://unitpp.sourceforge.net

Test::More

Test::More

Original author(s)	Michael G Schwern
Initial release	April, 2001
Development status	Active
Written in	Perl
Operating system	Cross-platform
Available in	English
Type	Unit testing module
License	Dual-licensed Artistic License and GPL

Test::More is a unit testing module for Perl. Created and maintained by Michael G Schwern with help from Barrie Slaymaker, Tony Bowden, chromatic, Fergal Daly and perl-qa. Introduced in 2001 to replace Test.pm, Test::More simplified and re-energized the culture of testing in Perl leading to an explosion of new testing modules and a strongly test driven community.

Test::More is the most popular Perl testing module, as of this writing about 80% of all CPAN distributions make use of it. Unlike other testing systems, Test::More is not a framework but can be used in concert with other testing libraries via a shared Test::Builder object. As a result, Test::More provides only the baseline testing functions leaving other libraries to implement more specific and sophisticated functionality. This removes what would otherwise be a development bottleneck and allows a rich eco-system of specialized niche testing functions.

Test::More is not a complete testing framework. Rather, test programs written with Test::More output their results as TAP which can then either be interpreted by a human, or more usually run through a TAP parser such as Test::Harness. It is this separation between test program and test result interpreter via a common protocol which allows Perl programmers to develop so many different testing modules and use them in combination. Additionally, the TAP output can be stored and reinterpreted later providing a historical record of test results.

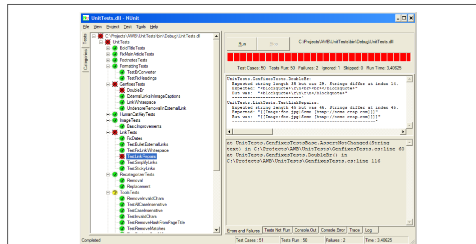
External links

- Test::More documentation [11]
- Test::More tutorial [1]

References

[1] http://search.cpan.org/perldoc?Test::Tutorial

NUnit

NUnit	
 NUnit 2.4.6 GUI on Windows	
Stable release	2.5.10 / April 2, 2011
Written in	C#
Operating system	Microsoft .NET, Mono
Type	Unit testing tool
License	BSD-style (modified zlib license)
Website	[1]

NUnit is an open source unit testing framework for Microsoft .NET. It serves the same purpose as JUnit does in the Java world, and is one of many in the xUnit family.

Features

Every test can be added to one or more categories, to allow running them selectively.[2]

Example

Example of an NUnit test fixture:

```
using NUnit.Framework;

 [TestFixture]
 public class ExampleTestOfNUnit
 {
   [Test]
   public void TestMultiplication()
   {
     Assert.AreEqual(4, 2*2, "Multiplication");

     // Equivalently, since version 2.4 NUnit offers a new and
     // more intuitive assertion syntax based on constraint objects
     // [http://www.nunit.org/index.php?p=constraintModel&r=2.4.7]:
     Assert.That(2*2, Is.EqualTo(4), "Multiplication constraint-based");
   }
 }
```

The NUnit framework discovers the method `ExampleTestOfNUnit.TestMultiplication()` automatically by reflection.

Extensions

FireBenchmarks[3] is an addin able to record execution time of unit tests and generate XML, CSV, XHTML performances reports with charts and history tracking. Its main purpose is to enable a developer or a team that work with an agile methodology to integrate performance metrics and analysis into the unit testing environment, to easily control and monitor the evolution of a software system in terms of algorithmic complexity and system resources load.

NUnit.Forms is an expansion to the core NUnit framework and is also open source. It specifically looks at expanding NUnit to be able to handle testing user interface elements in Windows Forms. As of August 2010, Nunit.Forms is in Alpha release, and no versions have been released since May 2006.

NUnit.ASP is a discontinued[4] expansion to the core NUnit framework and is also open source. It specifically looks at expanding NUnit to be able to handle testing user interface elements in ASP.NET.

References

[1] http://www.nunit.org
[2] "CategoryAttribute - NUnit documentation" (http://nunit.org/index.php?p=category&r=2.4.7). . Retrieved 2008-04-15.
[3] "Firebenchmarks website main page" (http://www.firebenchmarks.com/). .
[4] "NUnit.ASP website main page" (http://nunitasp.sourceforge.net/). Sourceforge. . Retrieved 2008-04-15.

Further reading

- Andrew Hunt, David Thomas: *Pragmatic Unit Testing in C# with NUnit, 2nd Ed.* The Pragmatic Bookshelf, Raleigh 2007, ISBN 0-9776166-7-3
- Jim Newkirk, Alexei Vorontsov: *Test-Driven Development in Microsoft .NET.* Microsoft Press, Redmond 2004, ISBN 0-7356-1948-4
- Bill Hamilton: *NUnit Pocket Reference*. O'Reilly, Cambridge 2004, ISBN 0-596-00739-6

External links

- Official website (http://http://www.nunit.org)
- Launchpad Site (https://launchpad.net/nunitv2)
- Test-driven Development with NUnit & Test-driven.NET (http://www.parlezuml.com/tutorials/tdd.html) video demonstration
- FireBenchmarks home page (http://www.firebenchmarks.com/)
- NUnit.Forms home page (http://nunitforms.sourceforge.net/)
- NUnitAsp homepage (http://nunitasp.sourceforge.net/)
- Article Improving Application Quality Using Test-Driven Development (http://www.methodsandtools.com/archive/archive.php?id=20) provides an introduction to TDD with concrete examples using Nunit

NUnitAsp

NUnitAsp is a tool for automatically testing ASP.NET web pages. It's an extension to NUnit, a tool for test-driven development in .NET.

How It Works

NUnitAsp is a class library for use within your NUnit tests. It provides NUnit with the ability to download, parse, and manipulate ASP.NET web pages.

With NUnitASP, your tests don't need to know how ASP.NET renders controls into HTML. Instead, you can rely on the NUnitASP library to do this for you, keeping your test code simple and clean. For example, your tests don't need to know that a DataGrid control renders as an HTML table. You can rely on NUnitASP to handle the details. This gives you the freedom to focus on functionality questions, like whether the DataGrid holds the expected values.

```
[Test]
public void TestExample()
{
  // First, instantiate "Tester" objects:
  LabelTester label = new LabelTester("textLabel", CurrentWebForm);
  LinkButtonTester link = new LinkButtonTester("linkButton", CurrentWebForm);
  // Second, visit the page being tested:
  Browser.GetPage("http://localhost/example/example.aspx");
  // Third, use tester objects to test the page:
  AssertEquals("Not clicked.", label.Text);
  link.Click();
  AssertEquals("Clicked once.", label.Text);
  link.Click();
  AssertEquals("Clicked twice.", label.Text);
}
```

NUnitAsp can test complex web sites involving multiple pages and nested controls.

Credits & History

NUnitAsp was created by Brian Knowles as a simple way to read and manipulate web documents with NUnit. Jim Shore (known at the time as "Jim Little") took over the project shortly afterwards and refactored it to the Tester-based approach used for the first release. Since then, more than a dozen people have contributed to the product. In November 2003, Levi Khatskevitch joined the team as "patch king" and brought new energy to the project, leading to the long-anticipated release of version 1.4. On January 31, 2008, Jim Shore announced the end of its development.

External links

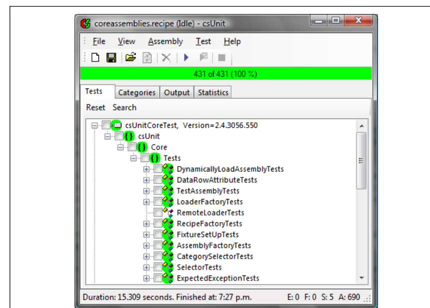
- NunitAsp Homepage [1]

References

[1] http://nunitasp.sourceforge.net/index.html

csUnit

csUnit



csUnit 2.5 GUI on Windows

Developer(s)	csUnit team
Stable release	csUnit 2.6 / March 28, 2009
Written in	C#
Operating system	Microsoft .NET
Type	Unit testing tool
License	zlib License
Website	[1]

csUnit is a unit testing framework for the .NET Framework. It is designed to work with any .NET compliant language. It has specifically been tested with C#, Visual Basic .NET, Managed C++, and J#. csUnit is open source and comes with a flexible license that allows cost-free inclusion in commercial closed-source products as well.

csUnit follows the concepts of other unit testing frameworks in the xUnit family and has had several releases since 2002. The tool offers a native GUI application, a command line, and addins for Visual Studio 2005 and Visual Studio 2008.

Starting with version 2.4 it also supports execution of NUnit tests without recompiling. This feature works for NUnit 2.4.7 (.NET 2.0 version).

csUnit supports .NET 3.5 and earlier versions, but does not support .NET 4.

csUnit has been integrated with ReSharper.

Special features

Along with the standard features, csUnit offers abilities that are uncommon in other unit testing frameworks for .NET:

- Categories to group included, excluded tests
- ExpectedException working with concrete instances rather than type only
- Out of the box addins for Visual Studio 2005 and 2008
- A tab for simple performance base lining
- A very rich set of assertions, continuously expanded
- Rich set of attributes for implementing tests
- Parameterized testing, data-driven testing
- Search abilities, saving time when test suites have thousands of tests

External links

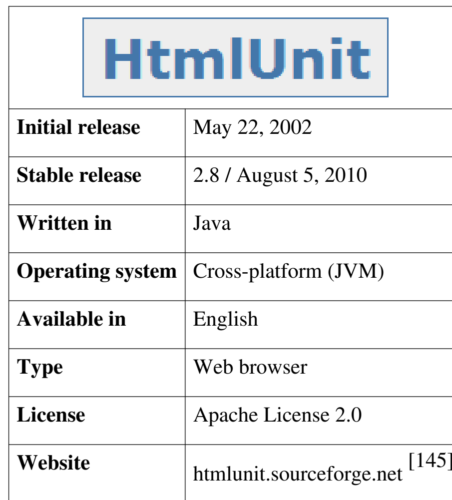
- [1]
- SourceForge Site [2]

References

[1] http://www.csunit.org
[2] http://sourceforge.net/projects/csunit

HtmlUnit

HtmlUnit



Initial release	May 22, 2002
Stable release	2.8 / August 5, 2010
Written in	Java
Operating system	Cross-platform (JVM)
Available in	English
Type	Web browser
License	Apache License 2.0
Website	htmlunit.sourceforge.net [145]

HtmlUnit is a headless web browser written in Java. It allows high-level manipulation of websites from other Java code, including filling and submitting forms and clicking hyperlinks. It also provides access to the structure and the details within received web pages. HtmlUnit emulates parts of browser behaviour including the lower-level aspects of TCP/IP and HTTP. A sequence such as `getPage(url)`, `getLinkWith("Click here")`, `click()` allows a user to navigate through hypertext and obtain web pages that include HTML, JavaScript, Ajax and cookies. This headless browser can deal with HTTPS security, basic http authentication, automatic page redirection and other HTTP headers. It allows Java test code to examine returned pages either as text, an XML DOM, or as collections of forms, tables, and links.[1]

The most common use of HtmlUnit is test automation of web pages, but sometimes it can be used for web scraping, or downloading website content.

Version 2.0 includes many new enhancements such as a W3C DOM implementation, Java 5 features, better XPath support, and improved handling for incorrect HTML, in addition to various JavaScript enhancements, while version 2.1 mainly focuses on tuning some performance issues reported by users.

References

[1] "HtmlUnit Home" (http://htmlunit.sourceforge.net/). . Retrieved 23 December 2010.

External links

- HtmlUnit (http://htmlunit.sourceforge.net/)

Test automation

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 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 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 though 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 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

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

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.

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)

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.

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)

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.

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.

Testing process

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

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]

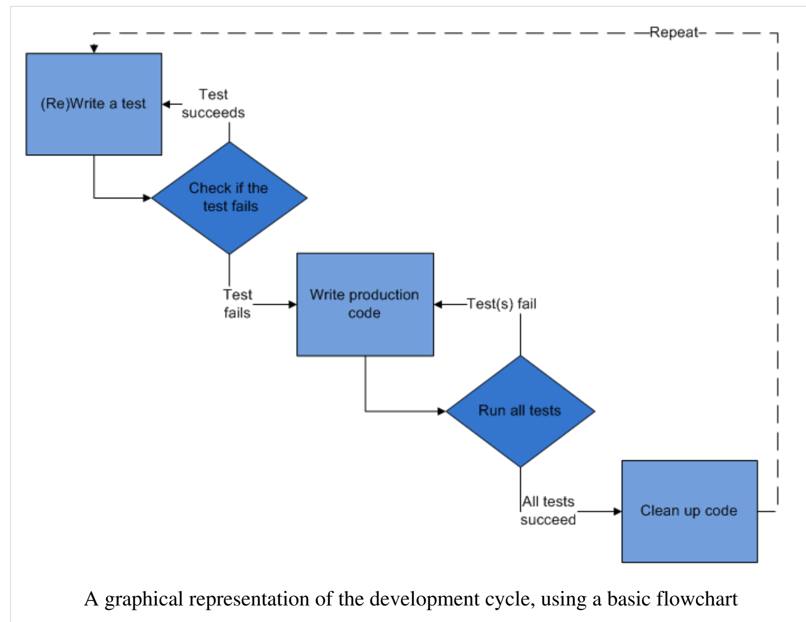
Programmers also apply the concept to improving and debugging legacy code developed with older techniques.[4]

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]



A graphical representation of the development cycle, using a basic flowchart

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). .

[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)
- 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)

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

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.

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.

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 a 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.

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=2HN1TRYZ4JGVAQSNDLRSKH0CJUNN2JVN). *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)

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/

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 8×5=40, but 142135 does not end in "0" like "40", while the correct answer does: 918×155=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.

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.

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

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.

Acceptance testing

Acceptance testing of an aircraft catapult

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.

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/)

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.testingeducation.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)

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 offer 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

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.

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 Veendendal. (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

Testing artefacts

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

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 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)

Traceability matrix

A **traceability matrix** is a document, usually in the form of a table, that correlates any two baselined documents that require a many to many relationship to determine the completeness of the relationship. It is often used with high-level requirements (these often consist of marketing requirements) and detailed requirements of the software product to the matching parts of high-level design, detailed design, test plan, and test cases.

A **requirements traceability matrix** may be used to check to see if the current project requirements are being met, and to help in the creation of a Request for Proposal, various deliverable documents, and project plan tasks.[1]

Common usage is to take the identifier for each of the items of one document and place them in the left column. The identifiers for the other document are placed across the top row. When an item in the left column is related to an item across the top, a mark is placed in the intersecting cell. The number of relationships are added up for each row and each column. This value indicates the mapping of the two items. Zero values indicate that no relationship exists. It must be determined if one must be made. Large values imply that the relationship is too complex and should be simplified.

To ease the creation of traceability matrices, it is advisable to add the relationships to the source documents for both backward traceability and forward traceability. In other words, when an item is changed in one baselined document, it's easy to see what needs to be changed in the other.

Sample Traceability matrix

Requirement Identifiers	Reqs Tested	REQ1 UC 1.1	REQ1 UC 1.2	REQ1 UC 1.3	REQ1 UC 2.1	REQ1 UC 2.2	REQ1 UC 2.3.1	REQ1 UC 2.3.2	REQ1 UC 2.3.3	REQ1 UC 2.4	REQ1 UC 3.1	REQ1 UC 3.2	REQ1 TECH 1.1	REQ1 TECH 1.2	REQ1 TECH 1.3
Test Cases	321	3	2	3	1	1	1	1	1	1	2	3	1	1	1
Tested Implicitly	77														
1.1.1	1	x													
1.1.2	2		x	x											
1.1.3	2	x											x		
1.1.4	1			x											
1.1.5	2	x												x	
1.1.6	1		x												
1.1.7	1			x											
1.2.1	2				x		x								
1.2.2	2					x		x							
1.2.3	2								x	x					
1.3.1	1										x				
1.3.2	1										x				
1.3.3	1											x			
1.3.4	1											x			
1.3.5	1											x			
etc…															
5.6.2	1														x

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-Kannenberg.pdf) by Andrew Kannenberg and Dr. Hossein Saiedian

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 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 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* in 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 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.

Static testing

Static testing

Static testing is a form of software testing where the software isn't actually used. This is in contrast to dynamic testing. It is generally not detailed testing, but checks mainly for the sanity of the code, algorithm, or document. It is primarily syntax checking of the code and/or manually reviewing the code or document to find errors. This type of testing can be used by the developer who wrote the code, in isolation. Code reviews, inspections and walkthroughs are also used.

From the black box testing point of view, static testing involves reviewing requirements and specifications. This is done with an eye toward completeness or appropriateness for the task at hand. This is the **verification** portion of Verification and Validation.

Even static testing can be automated. A static testing test suite consists of programs to be analyzed by an interpreter or a compiler that asserts the programs syntactic validity.

Bugs discovered at this stage of development are less expensive to fix than later in the development cycle.

The people involved in static testing are application developers and testers.

Sources

- Kaner, Cem; Nguyen, Hung Q; Falk, Jack (1988). *Testing Computer Software* (Second ed.). Boston: Thomson Computer Press. ISBN 0-47135-846-0.
- Static Testing C++ Code: A utility to check library usability [1]

References

[1] http://www.ddj.com/cpp/205801074

Software review

A **software review** is "A process or meeting during which a software product is examined by a project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval".[1]

In this context, the term "software product" means "any technical document or partial document, produced as a deliverable of a software development activity", and may include documents such as contracts, project plans and budgets, requirements documents, specifications, designs, source code, user documentation, support and maintenance documentation, test plans, test specifications, standards, and any other type of specialist work product.

Varieties of software review

Software reviews may be divided into three categories:

- Software peer reviews are conducted by the author of the work product, or by one or more colleagues of the author, to evaluate the technical content and/or quality of the work.[2]
- Software management reviews are conducted by management representatives to evaluate the status of work done and to make decisions regarding downstream activities.
- Software audit reviews are conducted by personnel external to the software project, to evaluate compliance with specifications, standards, contractual agreements, or other criteria.

Different types of reviews

- Code review is systematic examination (often as peer review) of computer source code.
- Pair programming is a type of code review where two persons develop code together at the same workstation.
- Inspection is a very formal type of peer review where the reviewers are following a well-defined process to find defects.
- Walkthrough is a form of peer review where the author leads members of the development team and other interested parties through a software product and the participants ask questions and make comments about defects.
- Technical review is a form of peer review in which a team of qualified personnel examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards.

Formal versus informal reviews

"Formality" identifies the degree to which an activity is governed by agreed (written) rules. Software review processes exist across a spectrum of formality, with relatively unstructured activities such as "buddy checking" towards one end of the spectrum, and more formal approaches such as walkthroughs, technical reviews, and software inspections, at the other. IEEE Std. 1028-1997 defines formal structures, roles, and processes for each of the last three ("formal peer reviews"), together with software audits.[1]

Research studies tend to support the conclusion that formal reviews greatly outperform informal reviews in cost-effectiveness. Informal reviews may often be unnecessarily expensive (because of time-wasting through lack of focus), and frequently provide a sense of security which is quite unjustified by the relatively small number of real defects found and repaired.

IEEE 1028 generic process for formal reviews

IEEE Std 1028 defines a common set of activities for "formal" reviews (with some variations, especially for software audit). The sequence of activities is largely based on the software inspection process originally developed at IBM by Michael Fagan.[3] Differing types of review may apply this structure with varying degrees of rigour, but all activities are mandatory for inspection:

- **0. [Entry evaluation]:** The Review Leader uses a standard checklist of entry criteria to ensure that optimum conditions exist for a successful review.
- **1. Management preparation:** Responsible management ensure that the review will be appropriately resourced with staff, time, materials, and tools, and will be conducted according to policies, standards, or other relevant criteria.
- **2. Planning the review:** The Review Leader identifies or confirms the objectives of the review, organises a team of Reviewers, and ensures that the team is equipped with all necessary resources for conducting the review.
- **3. Overview of review procedures:** The Review Leader, or some other qualified person, ensures (at a meeting if necessary) that all Reviewers understand the review goals, the review procedures, the materials available to them, and the procedures for conducting the review.
- **4. [Individual] Preparation:** The Reviewers individually prepare for group examination of the work under review, by examining it carefully for *anomalies* (potential defects), the nature of which will vary with the type of review and its goals.
- **5. [Group] Examination:** The Reviewers meet at a planned time to pool the results of their preparation activity and arrive at a consensus regarding the status of the document (or activity) being reviewed.
- **6. Rework/follow-up:** The Author of the work product (or other assigned person) undertakes whatever actions are necessary to repair defects or otherwise satisfy the requirements agreed to at the Examination meeting. The Review Leader verifies that all action items are closed.
- **7. [Exit evaluation]:** The Review Leader verifies that all activities necessary for successful review have been accomplished, and that all outputs appropriate to the type of review have been finalised.

Value of reviews

The most obvious value of software reviews (especially formal reviews) is that they can identify issues earlier and more cheaply than they would be identified by testing or by field use (the **defect detection process**). The cost to find and fix a defect by a well-conducted review may be one or two orders of magnitude less than when the same defect is found by test execution or in the field.

A second, but ultimately more important, value of software reviews is that they can be used to train technical authors in the development of extremely low-defect documents, and also to identify and remove process inadequacies that encourage defects (the **defect prevention process**).

This is particularly the case for peer reviews if they are conducted early and often, on samples of work, rather than waiting until the work has been completed. Early and frequent reviews of small work samples can identify systematic errors in the Author's work processes, which can be corrected before further faulty work is done. This improvement in Author skills can dramatically reduce the time it takes to develop a high-quality technical document, and dramatically decrease the error-rate in using the document in downstream processes.

As a general principle, the earlier a technical document is produced, the greater will be the impact of its defects on any downstream activities and their work products. Accordingly, greatest value will accrue from early reviews of documents such as marketing plans, contracts, project plans and schedules, and requirements specifications. Researchers and practitioners have shown the effectiveness of reviewing process in finding bugs and security issues,[4] .

References

[1] IEEE Std. 1028-1997, "IEEE Standard for Software Reviews", clause 3.5
[2] Wiegers, Karl E. (2001). *Peer Reviews in Software: A Practical Guide* (http://books.google.com/books?id=d7BQAAAAMAAJ&pgis=1). Addison-Wesley. p. 14. ISBN 0201734850. .
[3] Fagan, Michael E: "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, Vol. 15, No. 3, 1976; "Inspecting Software Designs and Code", *Datamation*, October 1977; "Advances In Software Inspections", *IEEE Transactions in Software Engineering*, Vol. 12, No. 7, July 1986
[4] Charles P.Pfleeger, Shari Lawrence Pfleeger. Security in Computing. Fourth edition. ISBN 0-13-239077-9

Software peer review

In software development, peer review is a type of software review in which a work product (document, code, or other) is examined by its author and one or more colleagues, in order to evaluate its technical content and quality.

Purpose

The purpose of a peer review is to provide "a disciplined engineering practice for detecting and correcting defects in software artifacts, and preventing their leakage into field operations" according to the Capability Maturity Model.

When performed as part of each Software development process activity, peer reviews identify problems that can be fixed early in the lifecycle.[1] That is to say, a peer review that identifies a requirements problem during the Requirements analysis activity is cheaper and easier to fix than during the Software architecture or Software testing activities.

The National Software Quality Experiment,[2] evaluating the effectiveness of peer reviews, finds, "a favorable return on investment for software inspections; savings exceeds costs by 4 to 1". To state it another way, it is four times more costly, on average, to identify and fix a software problem later.

Distinction from other types of software review

Peer reviews are distinct from management reviews, which are conducted by management representatives rather than by colleagues, and for management and control purposes rather than for technical evaluation. They are also distinct from software audit reviews, which are conducted by personnel external to the project, to evaluate compliance with specifications, standards, contractual agreements, or other criteria.

Review processes

Peer review processes exist across a spectrum of formality, with relatively unstructured activities such as "buddy checking" towards one end of the spectrum, and more formal approaches such as walkthroughs, technical peer reviews, and software inspections, at the other. The IEEE defines formal structures, roles, and processes for each of the last three.[3]

Management representatives are typically not involved in the conduct of a peer review except when included because of specific technical expertise or when the work product under review is a management-level document. This is especially true of line managers of other participants in the review.

Processes for formal peer reviews, such as software inspections, define specific roles for each participant, quantify stages with entry/exit criteria, capture software metrics on the peer review process.

"Open source" reviews

In the free / open source community, something like peer review has taken place in the engineering and evaluation of computer software. In this context, the rationale for peer review has its equivalent in Linus's law, often phrased: "Given enough eyeballs, all bugs are shallow", meaning "If there are enough reviewers, all problems are easy to solve." Eric S. Raymond has written influentially about peer review in software development.[4]

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. 261. ISBN 0470042125. .
[2] National Software Quality Experiment Resources and Results (http://members.aol.com/ONeillDon/nsqe-results.html)
[3] IEEE Std. 1028-2008, "IEEE Standard for Software Reviews and Audits" (http://ieeexplore.ieee.org/servlet/opac?punumber=4601582)
[4] Eric S. Raymond. *The Cathedral and the Bazaar*.

Software audit review

A **software audit review**, or software audit, is a type of software review in which one or more auditors who are not members of the software development organization conduct "An independent examination of a software product, software process, or set of software processes to assess compliance with specifications, standards, contractual agreements, or other criteria" [1] .

"Software product" mostly, but not exclusively, refers to some kind of technical document. IEEE Std. 1028 offers a list of 32 "examples of software products subject to audit", including documentary products such as various sorts of plan, contracts, specifications, designs, procedures, standards, and reports, but also non-documentary products such as data, test data, and deliverable media.

Software audits are distinct from software peer reviews and software management reviews in that they are conducted by personnel external to, and independent of, the software development organization, and are concerned with compliance of products or processes, rather than with their technical content, technical quality, or managerial implications.

The term "software audit review" is adopted here to designate the form of software audit described in IEEE Std. 1028.

Objectives and participants

"The purpose of a software audit is to provide an independent evaluation of conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures" [2] . The following roles are recommended:

- The *Initiator* (who might be a manager in the audited organization, a customer or user representative of the audited organization, or a third party), decides upon the need for an audit, establishes its purpose and scope, specifies the evaluation criteria, identifies the audit personnel, decides what follow-up actions will be required, and distributes the audit report.
- The *Lead Auditor* (who must be someone "free from bias and influence that could reduce his ability to make independent, objective evaluations") is responsible for administrative tasks such as preparing the audit plan and assembling and managing the audit team, and for ensuring that the audit meets its objectives.
- The *Recorder* documents anomalies, action items, decisions, and recommendations made by the audit team.
- The *Auditors* (who must be, like the Lead Auditor, free from bias) examine products defined in the audit plan, document their observations, and recommend corrective actions. (There may be only a single auditor.)

- The *Audited Organization* provides a liaison to the auditors, and provides all information requested by the auditors. When the audit is completed, the audited organization should implement corrective actions and recommendations.

Tools

Parts of Software audit could be done using static analysis tools that analyze application code and score its conformance with standards, guidelines, best practices. From the List of tools for static code analysis some are covering a very large spectrum from code to architecture review, and could be use for benchmarking.

References

[1] IEEE Std. 1028-1997, *IEEE Standard for Software Reviews*, clause 3.2
[2] IEEE Std. 10281997, clause 8.1

Software technical review

A **software technical review** is a form of peer review in which "a team of qualified personnel ... examines the suitability of the software product for its intended use and identifies discrepancies from specifications and standards. Technical reviews may also provide recommendations of alternatives and examination of various alternatives" (IEEE Std. 1028-1997, *IEEE Standard for Software Reviews*, clause 3.7).

"Software product" normally refers to some kind of technical document. This might be a software design document or program source code, but use cases, business process definitions, test case specifications, and a variety of other technical documentation, may also be subject to technical review.

Technical review differs from software walkthroughs in its specific focus on the technical quality of the product reviewed. It differs from software inspection in its ability to suggest direct alterations to the product reviewed, and its lack of a direct focus on training and process improvement.

The term **formal technical review** is sometimes used to mean a software inspection.

Objectives and participants

The purpose of a technical review is to arrive at a technically superior version of the work product reviewed, whether by correction of defects or by recommendation or introduction of alternative approaches. While the latter aspect may offer facilities that software inspection lacks, there may be a penalty in time lost to technical discussions or disputes which may be beyond the capacity of some participants.

IEEE 1028 recommends the inclusion of participants to fill the following roles:

The **Decision Maker** (the person for whom the technical review is conducted) determines if the review objectives have been met.

The **Review Leader** is responsible for performing administrative tasks relative to the review, ensuring orderly conduct, and ensuring that the review meets its objectives.

The **Recorder** documents anomalies, action items, decisions, and recommendations made by the review team.

Technical staff are active participants in the review and evaluation of the software product.

Management staff may participate for the purpose of identifying issues that require management resolution.

Customer or user representatives may fill roles determined by the Review Leader prior to the review.

A single participant may fill more than one role, as appropriate.

Process

A formal technical review will follow a series of activities similar to that specified in clause 5 of IEEE 1028, essentially summarised in the article on software review.

Management review

A **management review** is a management study into a project's status and allocation of resources. It is different from both a software engineering peer review, which evaluates the technical quality of software products, and a software audit, which is an externally conducted audit into a project's compliance to specifications, contractual agreements, and other criteria.

Process

A management review can be an informal process, but generally requires a formal structure and rules of conduct, such as those advocated in the IEEE standard, which are:[1]

1. Evaluate entry?
2. Management preparation?
3. Plan the structure of the review
4. Overview of review procedures?
5. [Individual] Preparation?
6. [Group] Examination?
7. Rework/follow-up?
8. [Exit evaluation]?

Definition

In software engineering, a **management review** is defined by the IEEE as:

> A systematic evaluation of a software acquisition, supply, development, operation, or maintenance process performed by or on behalf of management ... [and conducted] to monitor progress, determine the status of plans and schedules, confirm requirements and their system allocation, or evaluate the effectiveness of management approaches used to achieve fitness for purpose. Management reviews support decisions about corrective actions, changes in the allocation of resources, or changes to the scope of the project.
>
> Management reviews are carried out by, or on behalf of, the management personnel having direct responsibility for the system. Management reviews identify consistency with and deviations from plans, or adequacies and inadequacies of management procedures. This examination may require more than one meeting. The examination need not address all aspects of the product."[2]

References

[1] IEEE Std. 1028-1997, *IEEE Standard for Software Reviews*, clauses "Entry criteria"; 4.5, "Procedures"; 4.6, "Exit criteria"
[2] IEEE Std. 1028-1997, *IEEE Standard for Software Reviews*, clauses 3.4, 4.1

Software inspection

Inspection in software engineering, refers to peer review of any work product by trained individuals who look for defects using a well defined process. An inspection might also be referred to as a Fagan inspection after Michael Fagan, the creator of a very popular software inspection process.

Introduction

An inspection is one of the most common sorts of review practices found in software projects. The goal of the inspection is for all of the inspectors to reach consensus on a work product and approve it for use in the project. Commonly inspected work products include software requirements specifications and test plans. In an inspection, a work product is selected for review and a team is gathered for an inspection meeting to review the work product. A moderator is chosen to moderate the meeting. Each inspector prepares for the meeting by reading the work product and noting each defect. **The goal of the inspection is to identify defects**. In an inspection, a defect is any part of the work product that will keep an inspector from approving it. For example, if the team is inspecting a software requirements specification, each defect will be text in the document which an inspector disagrees with.

The process

The inspection process was developed by Michael Fagan in the mid-1970s and it has later been extended and modified.

The process should have entry criteria that determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process. The entry criteria might be a checklist including items such as "The document has been spell-checked".

The stages in the inspections process are: Planning, Overview meeting, Preparation, Inspection meeting, Rework and Follow-up. The Preparation, Inspection meeting and Rework stages might be iterated.

- **Planning:** The inspection is planned by the moderator.
- **Overview meeting:** The author describes the background of the work product.
- **Preparation:** Each inspector examines the work product to identify possible defects.
- **Inspection meeting:** During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.
- **Rework:** The author makes changes to the work product according to the action plans from the inspection meeting.
- **Follow-up:** The changes by the author are checked to make sure everything is correct.

The process is ended by the moderator when it satisfies some predefined exit criteria.

Inspection roles

During an inspection the following roles are used.

- **Author:** The person who created the work product being inspected.
- **Moderator:** This is the leader of the inspection. The moderator plans the inspection and coordinates it.
- **Reader:** The person reading through the documents, one item at a time. The other inspectors then point out defects.
- **Recorder/Scribe:** The person that documents the defects that are found during the inspection.
- **Inspector:** The person that examines the work product to identify possible defects.

Related inspection types

Code review

A code review can be done as a special kind of inspection in which the team examines a sample of code and fixes any defects in it. In a code review, a defect is a block of code which does not properly implement its requirements, which does not function as the programmer intended, or which is not incorrect but could be improved (for example, it could be made more readable or its performance could be improved). In addition to helping teams find and fix bugs, code reviews are useful for both cross-training programmers on the code being reviewed and for helping junior developers learn new programming techniques.

Peer Reviews

Peer reviews are considered an industry best-practice for detecting software defects early and learning about software artifacts. Peer Reviews are composed of software walkthroughs and software inspections and are integral to software product engineering activities. A collection of coordinated knowledge, skills, and behaviors facilitates the best possible practice of Peer Reviews. The elements of Peer Reviews include the structured review process, standard of excellence product checklists, defined roles of participants, and the forms and reports.

Software inspections are the most rigorous form of Peer Reviews and fully utilize these elements in detecting defects. Software walkthroughs draw selectively upon the elements in assisting the producer to obtain the deepest understanding of an artifact and reaching a consensus among participants. Measured results reveal that Peer Reviews produce an attractive return on investment obtained through accelerated learning and early defect detection. For best results, Peer Reviews are rolled out within an organization through a defined program of preparing a policy and procedure, training practitioners and managers, defining measurements and populating a database structure, and sustaining the roll out infrastructure.

External links

- Review and inspection practices [1]
- Article Software Inspections [2] by Ron Radice
- Comparison of different inspection and review techniques [3]
- Inspection Software [4] Information Portal

References

[1] http://www.stellman-greene.com/reviews
[2] http://www.methodsandtools.com/archive/archive.php?id=29
[3] http://www.the-software-experts.de/e_dta-sw-test-inspection.htm
[4] http://www.inspectionsoftware.info

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.

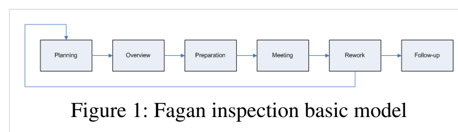


Figure 1: Fagan inspection basic model

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

Software walkthrough

In software engineering, a **walkthrough** or **walk-through** is a form of software peer review "in which a designer or programmer leads members of the development team and other interested parties through a software product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems"[1] .

"Software product" normally refers to some kind of technical document. As indicated by the IEEE definition, this might be a software design document or program source code, but use cases, business process definitions, test case specifications, and a variety of other technical documentation may also be walked through.

A walkthrough differs from software technical reviews in its openness of structure and its objective of familiarization. It differs from software inspection in its ability to suggest direct alterations to the product reviewed, its lack of a direct focus on training and process improvement, and its omission of process and product measurement.

Process

A walkthrough may be quite informal, or may follow the process detailed in IEEE 1028 and outlined in the article on software reviews.

Objectives and participants

In general, a walkthrough has one or two broad objectives: to gain feedback about the technical quality or content of the document; and/or to familiarize the audience with the content.

A walkthrough is normally organized and directed by the author of the technical document. Any combination of interested or technically qualified personnel (from within or outside the project) may be included as seems appropriate.

IEEE 1028[1] recommends three specialist roles in a walkthrough:

- The **Author**, who presents the software product in step-by-step manner at the walk-through meeting, and is probably responsible for completing most action items;

- The **Walkthrough Leader**, who conducts the walkthrough, handles administrative tasks, and ensures orderly conduct (and who is often the Author); and
- The **Recorder**, who notes all anomalies (potential defects), decisions, and action items identified during the walkthrough meetings.

References

[1] IEEE Std. 1028-1997, *IEEE Standard for Software Reviews*, clause 3.8

Code review

Code review is systematic examination (often as peer review) of computer source code. It is intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills. Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.[1]

Introduction

Code reviews can often find and remove common vulnerabilities such as format string exploits, race conditions, memory leaks and buffer overflows, thereby improving software security. Online software repositories based on Subversion (with Redmine or Trac), Mercurial, Git or others allow groups of individuals to collaboratively review code. Additionally, specific tools for collaborative code review can facilitate the code review process.

Automated code reviewing software lessens the task of reviewing large chunks of code on the developer by systematically checking source code for known vulnerabilities.

Capers Jones' ongoing analysis of over 12,000 software development projects showed that the latent defect discovery rate of formal inspection is in the 60-65% range. For informal inspection, the figure is less than 50%. The latent defect discovery rate for most forms of testing is about 30%. [2]

Typical code review rates are about 150 lines of code per hour. Inspecting and reviewing more than a few hundred lines of code per hour for critical software (such as safety critical embedded software) may be too fast to find errors. [3] Industry data indicate that code review can accomplish at most an 85% defect removal rate with an average rate of about 65%. [4]

Types

Code review practices fall into three main categories: pair programming, formal code review and lightweight code review.[1]

Formal code review, such as a Fagan inspection, involves a careful and detailed process with multiple participants and multiple phases. Formal code reviews are the traditional method of review, in which software developers attend a series of meetings and review code line by line, usually using printed copies of the material. Formal inspections are extremely thorough and have been proven effective at finding defects in the code under review.

Lightweight code review typically requires less overhead than formal code inspections, though it can be equally effective when done properly. Lightweight reviews are often conducted as part of the normal development process:

- Over-the-shoulder – One developer looks over the author's shoulder as the latter walks through the code.
- Email pass-around – Source code management system emails code to reviewers automatically after checkin is made.
- Pair Programming – Two authors develop code together at the same workstation, such is common in Extreme Programming.

- Tool-assisted code review – Authors and reviewers use specialized tools designed for peer code review.

Some of these may also be labeled a "Walkthrough" (informal) or "Critique" (fast and informal).

Many teams that eschew traditional, formal code review use one of the above forms of lightweight review as part of their normal development process. A code review case study published in the book *Best Kept Secrets of Peer Code Review* found that lightweight reviews uncovered as many bugs as formal reviews, but were faster and more cost-effective.

Criticism

Historically, formal code reviews have required a considerable investment in preparation for the review event and execution time.

Some believe that skillful, disciplined use of a number of other development practices can result in similarly high latent defect discovery/avoidance rates. Further, XP (extreme programming) proponents might argue, layering additional XP practices, such as refactoring and test-driven development will result in latent defect levels rivaling those achievable with more traditional approaches, without the investment.

Use of code analysis tools can support this activity. Especially tools that work in the IDE as they provide direct feedback to developers of coding standard compliance.

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. 260. ISBN 0470042125. .

[2] Jones, Capers; Christof, Ebert (April 2009). "Embedded Software: Facts, Figures, and Future" (http://doi.ieeecomputersociety.org/10.1109/MC.2009.118). IEEE Computer Society. . Retrieved 2010-10-05.

[3] Ganssle, Jack (February 2010). "A Guide to Code Inspections" (http://www.ganssle.com/inspections.pdf). The Ganssle Group. . Retrieved 2010-10-05.

[4] Jones, Capers (June 2008). "Measuring Defect Potentials and Defect Removal Efficiency" (http://www.stsc.hill.af.mil/crosstalk/2008/06/0806jones.html). Crosstalk, The Journal of Defense Software Engineering. . Retrieved 2010-10-05.

- Jason Cohen (2006). *Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.)*. Smartbearsoftware.com. ISBN 1599160676.

External links

- *Security Code Review FAQs* (http://www.ouncelabs.com/resources/code-review-faq.asp)
- Security code review guidelines (http://www.homeport.org/~adam/review.html)
- Lightweight Tool Support for Effective Code Reviews (http://www.atlassian.com/software/crucible/learn/codereviewwhitepaper.pdf) white paper
- Code Review Best Practices (http://www.parasoft.com/jsp/printables/When_Why_How_Code_Review.pdf?path=/jsp/products/article.jsp) white paper by Adam Kolawa
- Best Practices for Peer Code Review (http://smartbear.com/docs/BestPracticesForPeerCodeReview.pdf) white paper
- Code review case study (http://smartbear.com/resources/cc/CaseStudy-Cisco.pdf)
- "A Guide to Code Inspections" (Jack G. Ganssle) (http://www.ganssle.com/inspections.pdf)
- Article Four Ways to a Practical Code Review (http://www.methodsandtools.com/archive/archive.php?id=66)
- http://www.reviewboard.org/
- http://www.agilereview.org

Automated code review

Automated code review software checks source code for compliance with a predefined set of rules or best practices. The use of analytical methods to inspect and review source code to detect bugs has been a standard development practice. This process can be accomplished both manually and in an automated fashion[1] . With automation, software tools provide assistance with the code review and inspection process. The review program or tool typically displays a list of warnings (violations of programming standards). A review program can also provide an automated or a programmer-assisted way to correct the issues found.

Some static code analysis tools can be used to assist with automated code review. They compare favorably to manual reviews, but they can be done faster and more efficiently. These tools also encapsulate deep knowledge of underlying rules and semantics required to perform this type analysis such that it does not require the human code reviewer to have the same level of expertise as an expert human auditor[1] . Many Integrated Development Environments also provide basic automated code review functionality. For example the Eclipse[2] and Microsoft Visual Studio[3] IDEs support a variety of plugins that facilitate code review.

Next to static code analysis tools, there are also tools that analyze and visualize software structures and help humans to better understand these. Such systems are geared more to analysis because they typically do not contain a predefined set of rules to check software against. Some of these tools (e.g. SonarJ, Sotoarc, Structure101) allow to define target architectures and enforce that target architecture constraints are not violated by the actual software implementation.

References

[1] Gomes, Ivo; Morgado, Pedro; Gomes, Tiago; Moreira, Rodrigo (2009). "An overview of the Static Code Analysis approach in Software Development" (http://paginas.fe.up.pt/~ei05021/TQSO - An overview on the Static Code Analysis approach in Software Development. pdf). Universadide do Porto. . Retrieved 2010-10-03.

[2] "Collaborative Code Review Tool Development" (http://marketplace.eclipse.org/content/collaborative-code-review-tool). www.eclipse.org. . Retrieved 2010-10-13.

[3] "Code Review Plug-in for Visual Studio 2008, ReviewPal" (http://www.codeproject.com/KB/work/ReviewPal.aspx). www.codeproject.com. . Retrieved 2010-10-13.

Code reviewing software

Automated code review software checks source code for compliance with a predefined set of rules or best practices. The use of analytical methods to inspect and review source code to detect bugs has been a standard development practice. This process can be accomplished both manually and in an automated fashion[1] . With automation, software tools provide assistance with the code review and inspection process. The review program or tool typically displays a list of warnings (violations of programming standards). A review program can also provide an automated or a programmer-assisted way to correct the issues found.

Some static code analysis tools can be used to assist with automated code review. They compare favorably to manual reviews, but they can be done faster and more efficiently. These tools also encapsulate deep knowledge of underlying rules and semantics required to perform this type analysis such that it does not require the human code reviewer to have the same level of expertise as an expert human auditor[1] . Many Integrated Development Environments also provide basic automated code review functionality. For example the Eclipse[2] and Microsoft Visual Studio[3] IDEs support a variety of plugins that facilitate code review.

Next to static code analysis tools, there are also tools that analyze and visualize software structures and help humans to better understand these. Such systems are geared more to analysis because they typically do not contain a predefined set of rules to check software against. Some of these tools (e.g. SonarJ, Sotoarc, Structure101) allow to define target architectures and enforce that target architecture constraints are not violated by the actual software implementation.

References

[1] Gomes, Ivo; Morgado, Pedro; Gomes, Tiago; Moreira, Rodrigo (2009). "An overview of the Static Code Analysis approach in Software Development" (http://paginas.fe.up.pt/~ei05021/TQSO - An overview on the Static Code Analysis approach in Software Development. pdf). Universadide do Porto. . Retrieved 2010-10-03.
[2] "Collaborative Code Review Tool Development" (http://marketplace.eclipse.org/content/collaborative-code-review-tool). www.eclipse.org. . Retrieved 2010-10-13.
[3] "Code Review Plug-in for Visual Studio 2008, ReviewPal" (http://www.codeproject.com/KB/work/ReviewPal.aspx). www.codeproject.com. . Retrieved 2010-10-13.

Static code analysis

Static program 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)[1] In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension or code review.

The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

It can be argued that software metrics and reverse engineering are forms of static analysis. In fact deriving software metrics and **static analysis** are increasingly deployed together, especially in creation of embedded systems, by defining so called *software quality objectives*[2] .

A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code[3] . For example the following industries have identified the use of static code analysis as a means of improving the quality of increasingly sophisticated and complex software:

1. Medical software: In the U.S. Food and Drug Administration (FDA) has identified the use of static analysis for medical devices.[4]
2. Nuclear software: In the UK the Health and Safety Executive recommends the use of Static Analysis on Reactor Protection Systems.[5]

Formal methods

Formal methods is the term applied to the analysis of software (and hardware) whose results are obtained purely through the use of rigorous mathematical methods. The mathematical techniques used include denotational semantics, axiomatic semantics, operational semantics, and abstract interpretation.

By a straightforward reduction to the halting problem it is possible to prove that (for any Turing complete language) finding all possible run-time errors in an arbitrary program (or more generally any kind of violation of a specification on the final result of a program) is undecidable: there is no mechanical method that can always answer truthfully whether a given program may or may not exhibit runtime errors. This result dates from the works of Church, Gödel and Turing in the 1930s (see the halting problem and Rice's theorem). As with most undecidable questions, one can still attempt to give useful approximate solutions.

Some of the implementation techniques of formal static analysis include:

- Model checking considers systems that have finite state or may be reduced to finite state by abstraction;
- Data-flow analysis is a lattice-based technique for gathering information about the possible set of values;
- Abstract interpretation models the effect that every statement has on the state of an abstract machine (i.e., it 'executes' the software based on the mathematical properties of each statement and declaration). This abstract machine over-approximates the behaviours of the system: the abstract system is thus made simpler to analyze, at the expense of *incompleteness* (not every property true of the original system is true of the abstract system). If properly done, though, abstract interpretation is *sound* (every property true of the abstract system can be mapped to a true property of the original system).[6] The Frama-c framework and Polyspace heavily rely on abstract interpretation.

- Use of assertions in program code as first suggested by Hoare logic. There is tool support for some programming languages (e.g., the SPARK programming language (a subset of Ada) and the Java Modeling Language — JML — using ESC/Java and ESC/Java2, ANSI/ISO C Specification Language for the C language).

References

[1] 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

[2] Software Quality Objectives for Source Code. Proceedings *Embedded Real Time Software and Systems 2010 Conference*, ERTS2, Toulouse, France: Patrick Briand, Martin Brochet, Thierry Cambois, Emmanuel Coutenceau, Olivier Guetta, Daniel Mainberte, Frederic Mondot, Patrick Munier, Loic Noury, Philippe Spozio, Frederic Retailleau http://www.erts2010.org/Site/0ANDGY78/Fichier/PAPIERS%20ERTS%202010/ERTS2010_0035_final.pdf

[3] Improving Software Security with Precise Static and Runtime Analysis, Benjamin Livshits, section 7.3 “Static Techniques for Security,” Stanford doctoral thesis, 2006. http://research.microsoft.com/en-us/um/people/livshits/papers/pdf/thesis.pdf

[4] FDA (2010-09-08). "Infusion Pump Software Safety Research at FDA" (http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm). Food and Drug Administration. . Retrieved 2010-09-09.

[5] Computer based safety systems - technical guidance for assessing software aspects of digital computer based protection systems, http://www.hse.gov.uk/foi/internalops/nsd/tech_asst_guides/tast046app1.htm

[6] Jones, Paul (2010-02-09). "A Formal Methods-based verification approach to medical device software analysis" (http://embeddeddsp.embedded.com/design/opensource/222700533). Embedded Systems Design. . Retrieved 2010-09-09.

Bibliography

- Syllabus and readings (http://www.stanford.edu/class/cs295/) for Alex Aiken (http://theory.stanford.edu/~aiken/)’s Stanford CS295 course.
- Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, “ Using Static Analysis to Find Bugs (http://www2.computer.org/portal/web/csdl/doi/10.1109/MS.2008.130),” IEEE Software, vol. 25, no. 5, pp. 22-29, Sep./Oct. 2008, doi:10.1109/MS.2008.130
- Brian Chess, Jacob West (Fortify Software) (2007). *Secure Programming with Static Analysis*. Addison-Wesley. ISBN 978-0321424778.
- Flemming Nielson, Hanne R. Nielson, Chris Hankin (1999, corrected 2004). *Principles of Program Analysis*. Springer. ISBN 978-3540654100.
- “Abstract interpretation and static analysis,” (http://santos.cis.ksu.edu/schmidt/Escuela03/home.html) International Winter School on Semantics and Applications 2003, by David A. Schmidt (http://people.cis.ksu.edu/~schmidt/)

External links

- The SAMATE Project (http://samate.nist.gov), a resource for Automated Static Analysis tools
- Integrate static analysis into a software development process (http://www.embedded.com/shared/printableArticle.jhtml?articleID=193500830)
- Code Quality Improvement - Coding standards conformance checking (DDJ) (http://www.ddj.com/dept/debug/189401916)
- Episode 59: Static Code Analysis (http://www.se-radio.net/index.php?post_id=220531) Interview (Podcast) at *Software Engineering Radio*
- Implementing Automated Governance for Coding Standards (http://www.infoq.com/articles/governance-coding-standards) Explains why and how to integrate static code analysis into the build process
- What is Static Code Analysis? explanation in Hebrew (http://eswlab.com/info.asp?cid=637) (Hebrew: ?מהו ניתוח קוד סטטי)

List of tools for static code analysis

This is a list of tools for static code analysis.

Historical products

- Lint — The original static code analyzer of C code.
- NuMega Code Review – now part of Micro Focus DevPartner suite.

Open-source or Non-commercial products

Multi-language

- Moose — Moose started as a software analysis platform with many tools to manipulate, assess or visualize software. It can evolve to a more generic data analysis platform. Supported programming languages are Java, Smalltalk, C/C++, .Net. Other languages may be added.
- Copy/Paste Detector (CPD) — PMDs duplicate code detection for (e.g.) Java, JSP, C, C++ and PHP code.
- Sonar — A continuous inspection engine to manage the technical debt (unit tests, complexity, duplication, design, comments, coding standards and potential problems). Supported languages are Java, Flex, PHP, PL/SQL, Cobol and Visual Basic 6.
- Yasca — Yet Another Source Code Analyzer, a plugin-based framework for scanning arbitrary file types, with plugins for scanning C/C++, Java, JavaScript, ASP, PHP, HTML/CSS, ColdFusion, COBOL, and other file types. It integrates with other scanners, including FindBugs, JLint, PMD, and Pixy.

.NET (C#, VB.NET and all .NET compatible languages)

- FxCop — Free static analysis for Microsoft .NET programs that compile to CIL. Standalone and integrated in some Microsoft Visual Studio editions. From Microsoft.
- Gendarme — Open-source (MIT License) equivalent to FxCop created by the Mono project. Extensible rule-based tool to find problems in .NET applications and libraries, particularly those that contain code in ECMA CIL format.
- StyleCop — Analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Microsoft Visual Studio or integrated into an MSBuild project. Free download from Microsoft.

ActionScript

- Apparat — A language manipulation and optimization framework consisting of intermediate representations for ActionScript.

C

- Antic — C and CPP analyzer, can detect errors such as division by 0 and array index bounds. A part of JLint, but can be used as standalone.
- BLAST — (Berkeley Lazy Abstraction Software verification Tool) — A software model checker for C programs based on lazy abstraction.
- Clang — A compiler that includes a static analyzer.
- Coccinelle — Source code pattern matching and transformation
- Frama-C — A static analysis framework for C.
- Lint — The original static code analyzer for C.
- Sparse — A tool designed to find faults in the Linux kernel.

- Splint — An open source evolved version of Lint (for C).

C++

- Clang — A compiler that includes a static analyzer.
- cppcheck — Open-source tool that checks for several types of errors, including the use of STL.
- cpplint [1] Open source, automated checker to make sure a C++ file follows Google's C++ style.
- Eclipse (software) — An IDE that includes a static code analyzer (CODAN).

Java

- Checkstyle — Besides some static code analysis, it can be used to show violations of a configured coding standard.
- FindBugs — An open-source static bytecode analyzer for Java (based on Jakarta BCEL) from the University of Maryland.
- Hammurapi — (Free for non-commercial use only) versatile code review solution.
- PMD — A static ruleset based Java source code analyzer that identifies potential problems.
- Soot — A language manipulation and optimization framework consisting of intermediate languages for Java.
- Squale — A platform to manage software quality (also available for other languages, using commercial analysis tools though).

JavaScript

- Closure Compiler — JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It also checks your usage of native javascript functions.
- JSLint — JavaScript syntax checker and validator.
- JSHint [2] — JavaScript syntax checker and validator. A fork of JSLint. Compatible with more javascript environments.

Objective-C

- Clang — The free Clang project includes a static analyzer. As of version 3.2, this analyzer is included in Xcode.[3]

CSS

- CSS Lint A tool to help point out problems with CSS code. [4]

Commercial products

Multi-language

- Axivion Bauhaus Suite — A tool for C, C++, C#, Java and Ada code that comprises various analyses such as architecture checking, interface analyses, and clone detection.
- Black Duck Suite — Analyze the composition of software source code and binary files, search for reusable code, manage open source and third-party code approval, honor the legal obligations associated with mixed-origin code, and monitor related security vulnerabilities.
- BugScout — Detects security flaws in Java, PHP, ASP and C# web applications.
- CAST Application Intelligence Platform — Detailed, audience-specific dashboards to measure quality and productivity. 30+ languages, SAP, Oracle, PeopleSoft, Siebel, .NET, Java, C/C++, Struts, Spring, Hibernate and all major databases.

- Checkmarx — Finds security vulnerabilities in source code, by analyzing source files and storing their meta data in a queryable database. Supports .NET family (C#, VB.Net), Java, C\C++, VB6, Apex+VF, PHP, ASP.
- Coverity Static Analysis (formerly Coverity Prevent) — Identifies security vulnerabilities and code defects in C, C++, C# and Java code. Complements Coverity Dynamic Code Analysis and Architecture Analysis.
- DevPartner Code Review. Offered by Micro Focus. Static metrics and bug pattern detection for C#, VB.NET, and ASP.NET languages. Plugin to Visual Studio. Customized parsers provide extension through regular expressions and tailored rulesets.
- DMS Software Reengineering Toolkit — Supports custom analysis of C, C++, C#, Java, COBOL, PHP, VisualBasic and many other languages. Also COTS tools for clone analysis, dead code analysis, and style checking.
- Compuware DevEnterprise — Analysis of COBOL, PL/I, JCL, CICS, DB2, IMS and others.
- GrammaTech CodeSonar — Analyzes C,C++.
- HP Fortify Source Code Analyzer — Helps developers identify software security vulnerabilities in C/C++, .NET, Java, JSP, ASP.NET, ColdFusion, "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL, python and COBOL as well as configuration files.
- Imagix 4D — Identifies problems in variable usage, task interaction and concurrency, particularly in embedded applications, as part of an overall solution for understanding, improving and documenting C, C++ and Java software.
- Intel - Intel Parallel Studio XE: Contains **Static Security Analysis** (SSA) feature supports C/C++ and Fortran
- JustCode — Code analysis and refactoring productivity tool for JavaScript, C#, Visual Basic.NET, and ASP.NET
- Klocwork Insight — Provides security vulnerability and defect detection as well as architectural and build-over-build trend analysis for C, C++, C# and Java.
- Lattix, Inc. LDM — Architecture and dependency analysis tool for Ada, C/C++, Java, .NET software systems.
- LDRA Testbed — A software analysis and testing tool suite for C, C++, Ada83, Ada95 and Assembler (Intel, Freescale, Texas Instruments).
- Logiscope[5] — Logiscope is a software quality assurance tool that automates code reviews and the identification and detection of error-prone modules for software testing.
- MALPAS; A software static analysis toolset for a variety of languages including Ada, C, Pascal and Assembler (Intel, PowerPC and Motorola). Used primarily for safety critical applications in Nuclear and Aerospace industries.
- Micro Focus (formerly Relativity Technologies) Modernization Workbench — Parsers included for COBOL (multiple variants including IBM, Unisys, MF, ICL, Tandem), PL/I, Natural (inc. ADABAS), Java, Visual Basic, RPG, C & C++ and other legacy languages; Extensible SDK to support 3rd party parsers. Supports automated Metrics (including Function Points), Business Rule Mining, Componentisation and SOA Analysis. Rich ad hoc diagramming, AST search & reporting)
- Ounce Labs (from 2010 IBM Rational Appscan Source) — Automated source code analysis that enables organizations to identify and eliminate software security vulnerabilities in languages including Java, JSP, C/C++, C#, ASP.NET and VB.Net.
- Parasoft — Analyzes Java (Jtest), JSP, C, C++ (C++test), .NET (C#, ASP.NET, VB.NET, etc.) using .TEST, WSDL, XML, HTML, CSS, JavaScript, VBScript/ASP, and configuration files for security,[6] compliance,[7] and defect prevention.
- Polyspace — Uses abstract interpretation to detect and prove the absence of certain run-time errors in source code for C, C++, and Ada
- ProjectCodeMeter[8] — Warns on code quality issues such as insufficient commenting or complex code structure. Counts code metrics, gives cost & time estimations. Analyzes C, C++, C#, J#, Java, PHP, Objective C, JavaScript, UnrealEngine script, ActionScript, DigitalMars D.
- Rational Software Analyzer — Supports Java, C/C++ (and others available through extensions)

- ResourceMiner — Architecture down to details multipurpose analysis and metrics, develop own rules for masschange and generator development. Supports 30+ legacy and modern languages and all major databases.
- SofCheck Inspector — Provides static detection of logic errors, race conditions, and redundant code for Java and Ada. Provides automated extraction of pre/postconditions from code itself.
- Software Diagnostics [9] — Analyzes and integrates metrics from code (C, C++, C#, Java, ABAP, COBOL, …), executed applications (performance, test coverage, …), and repositories (SCM, Bug/Issue-Tracking, …). Results are presented to managers and team leaders as "software maps" [9] that reveal costly and risky parts of the code.
- Sotoarc/Sotograph — Architecture and quality in-depth analysis and monitoring for Java, C#, C and C++
- Syhunt Sandcat — Detects security flaws in PHP, Classic ASP and ASP.NET web applications.
- Understand — Analyzes C,C++, Java, Ada, Fortran, Jovial, Delphi, VHDL, HTML, CSS, PHP, and JavaScript — reverse engineering of source, code navigation, and metrics tool.
- Veracode — Finds security flaws in application binaries and bytecode without requiring source. Supported languages include C, C++, .NET (C#, C++/CLI, VB.NET, ASP.NET), Java, JSP, ColdFusion, and PHP, including mobile applications on the Windows Mobile, BlackBerry, and Android platforms.
- Visual Studio Team System — Analyzes C++,C# source codes. only available in team suite and development edition.

.NET

Products covering multiple .NET languages.

- CodeIt.Right — Combines Static Code Analysis and automatic Refactoring to best practices which allows automatically correct code errors and violations. Supports both C# and VB.NET.
- CodeRush — A plugin for Visual Studio, it addresses a multitude of shortcomings with the popular IDE. Including alerting users to violations of best practices by using static code analysis.
- Parasoft dotTEST — A static analysis, unit testing, and code review plugin for Visual Studio; works with programming languages that target the Microsoft .NET Framework and .NET Compact Framework, including C#, VB.NET, ASP.NET and Managed C++.
- JustCode — Add-on for Visual Studio 2005/2008/2010 for real-time, solution-wide code analysis for C#, VB.NET, ASP.NET, XAML, JavaScript, HTML and multi-language solutions.
- NDepend — Simplifies managing a complex .NET code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code. Integrates into Visual Studio.
- ReSharper — Add-on for Visual Studio 2003/2005/2008/2010 from the creators of IntelliJ IDEA, which also provides static code analysis for C#.
- Kalistick — Mixing from the Cloud: static code analysis with best practice tips and collaborative tools for Agile teams

Ada

- Ada-ASSURED — A tool that offers coding style checks, standards enforcement and pretty printing features.
- AdaCore CodePeer — Automated code review and bug finder for Ada programs that uses control-flow, data-flow, and other advanced static analysis techniques.
- LDRA Testbed — A software analysis and testing tool suite for Ada83/95.
- Polyspace — Uses abstract interpretation to detect and prove the absence of certain run-time errors in source code
- SofCheck Inspector — Provides static detection of logic errors, race conditions, and redundant code for Ada. Provides automated extraction of pre/postconditions from code itself.

C / C++

- Astrée; exhaustive search for runtime errors and assertion violations by abstract interpretation; tailored towards critical code (avionics)
- FlexeLint — A multiplatform version of PC-Lint.
- Green Hills Software DoubleCheck — A software analysis tool for C/C++.
- Intel - Intel Parallel Studio XE: Contains **Static Security Analysis** (SSA) feature
- LDRA Testbed — A software analysis and testing tool suite for C/C++.
- Monoidics INFER — A sound tool for C/C++ based on Separation Logic.
- Parasoft C/C++test— A C and C++ tool that provides static analysis, unit testing, code review, and runtime error detection; plugins are available for Visual Studio and Eclipse-based IDEs.
- PC-Lint — A software analysis tool for C/C++.
- Polyspace — Uses abstract interpretation to detect and prove the absence of certain run-time errors in source code
- PVS-Studio — A software analysis tool for C/C++/C++0x.
- QA-C (and QA-C++) — Deep static analysis of C/C++ for quality assurance and guideline enforcement.
- Red Lizard's Goanna — Static analysis for C/C++ in Eclipse and Visual Studio.
- CppDepend — Simplifies managing a complex C\C++ code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code. Integrates into Visual Studio.

Java

- Jtest — Testing and static code analysis product by Parasoft.
- LDRA Testbed — A software analysis and testing tool suite for Java.
- SemmleCode — Object oriented code queries for static program analysis.
- SonarJ — Monitors conformance of code to intended architecture, also computes a wide range of software metrics.
- Kalistick — A Cloud-based platform to manage and optimize code quality for Agile teams with DevOps spirit

Formal methods tools

Tools that use a formal methods approach to static analysis (e.g., using static program assertions):

- ESC/Java and ESC/Java2 — Based on Java Modeling Language, an enriched version of Java.
- MALPAS; A formal methods tool that uses directed graphs and regular algebra to prove that software under analysis correctly meets its mathematical specification.
- Polyspace — Uses abstract interpretation (a formal methods based technique[10]) to detect and prove the absence of certain run-time errors in source code for C, C++, and Ada
- SofCheck Inspector — Statically determines and documents pre- and postconditions for Java methods; statically checks preconditions at all call sites; also supports Ada.
- SPARK Toolset including the SPARK Examiner — Based on the SPARK programming language, a subset of Ada.

References

[1] http://code.google.com/p/google-styleguide/
[2] http://jshint.com/
[3] "Static Analysis in Xcode" (http://developer.apple.com/mac/library/featuredarticles/StaticAnalysis/index.html). Apple. . Retrieved 2009-09-03.
[4] "Static A" (http://csslint.net/about.html). .
[5] http://www-01.ibm.com/software/awdtools/logiscope/
[6] Parasoft Application Security Solution (http://www.parasoft.com/jsp/solutions/application_security_solution.jsp?itemId=322)
[7] Parasoft Compliance Solution (http://www.parasoft.com/jsp/solutions/compliance.jsp?itemId=339)
[8] Project Code Meter site (http://www.projectcodemeter.com)
[9] http://www.softwarediagnostics.com
[10] Cousot, Patrick (2007). "The Role of Abstract Interpretation in Formal Methods" (http://ieeexplore.ieee.org/Xplore/login.jsp?url=http://ieeexplore.ieee.org/iel5/4343908/4343909/04343930.pdf?arnumber=4343930&authDecision=-203). IEEE International Conference on Software Engineering and Formal Methods. . Retrieved 2010-11-08.

External links

- Java Static Checkers (http://www.dmoz.org//Computers/Programming/Languages/Java/Development_Tools/Performance_and_Testing/Static_Checkers/) at the Open Directory Project
- List of Java static code analysis plugins for Eclipse (http://www.eclipseplugincentral.com/Web_Links-index-req-viewcatlink-cid-14-orderby-rating.html)
- List of static source code analysis tools for C (http://www.spinroot.com/static/)
- List of Static Source Code Analysis Tools (https://www.cert.org/secure-coding/tools.html) at CERT
- SAMATE-Source Code Security Analyzers (http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html)
- SATE - Static Analysis Tool Exposition (http://samate.nist.gov/SATE.html)
- “A Comparison of Bug Finding Tools for Java” (http://www.cs.umd.edu/~jfoster/papers/issre04.pdf), by Nick Rutar, Christian Almazan, and Jeff Foster, University of Maryland. Compares Bandera, ESC/Java 2, FindBugs, JLint, and PMD.
- “Mini-review of Java Bug Finders” (http://www.oreillynet.com/digitalmedia/blog/2004/03/minireview_of_java_bug_finders.html), by Rick Jelliffe, O'Reilly Media.
- Parallel Lint (http://www.ddj.com/218000153), by Andrey Karpov
- Integrate static analysis into a software development process (http://www.embedded.com/shared/printableArticle.jhtml?articleID=193500830) Explains how one goes about integrating static analysis into a software development process

GUI testing and review

GUI software 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 rsp. 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.

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.

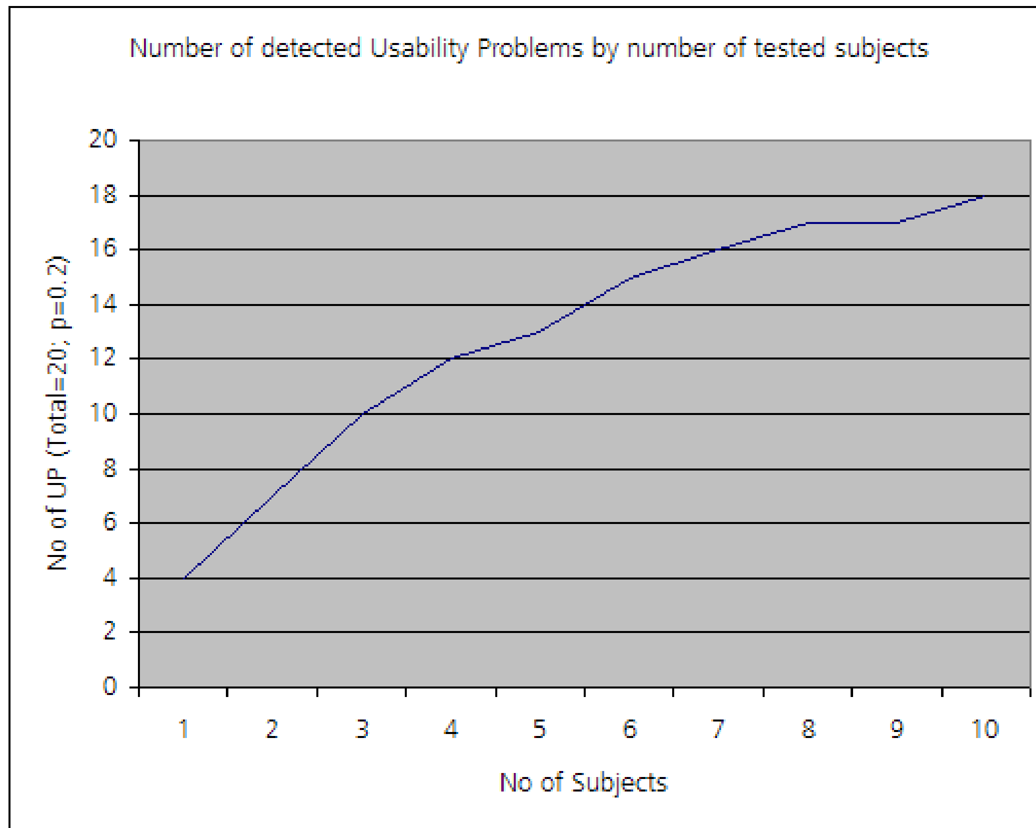
How many users to test?

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=lRs_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/000maxscrns.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)

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.

Usability inspection

Usability inspection is the name for a set of methods where an evaluator inspects a user interface. This is in contrast to usability testing where the usability of the interface is evaluated by testing it on real users. Usability inspections can generally be used early in the development process by evaluating prototypes or specifications for the system that can't be tested on users. Usability inspection methods are generally considered to be cheaper to implement than testing on users.[1]

Usability inspection methods include:

- Cognitive walkthrough (task-specific)
- Heuristic evaluation (holistic)
- Pluralistic walkthrough

References

[1] Nielsen, Jakob. Usability Inspection Methods. New York, NY: John Wiley and Sons, 1994

External links

- Summary of Usability Inspection Methods (http://www.useit.com/papers/heuristic/inspection_summary.html)

Cognitive walkthrough

The **cognitive walkthrough** method is a usability inspection method used to identify usability issues in a piece of software or web site, focusing on how easy it is for new users to accomplish tasks with the system. Whereas **cognitive walkthrough** is task-specific, heuristic evaluation takes a holistic view to catch problems not caught by this and other usability inspection methods. The method is rooted in the notion that users typically prefer to learn a system by using it to accomplish tasks, rather than, for example, studying a manual. The method is prized for its ability to generate results quickly with low cost, especially when compared to usability testing, as well as the ability to apply the method early in the design phases, before coding has even begun.

Introduction

A cognitive walkthrough starts with a task analysis that specifies the sequence of steps or actions required by a user to accomplish a task, and the system responses to those actions. The designers and developers of the software then walk through the steps as a group, asking themselves a set of questions at each step. Data is gathered during the walkthrough, and afterwards a report of potential issues is compiled. Finally the software is redesigned to address the issues identified.

The effectiveness of methods such as cognitive walkthroughs is hard to measure in applied settings, as there is very limited opportunity for controlled experiments while developing software. Typically measurements involve comparing the number of usability problems found by applying different methods. However, Gray and Salzman called into question the validity of those studies in their dramatic 1998 paper "Damaged Merchandise", demonstrating how very difficult it is to measure the effectiveness of usability inspection methods. However, the consensus in the usability community is that the cognitive walkthrough method works well in a variety of settings and applications.

Walking through the tasks

After the task analysis has been made the participants perform the walkthrough by asking themselves a set of questions for each subtask. Typically four questions are asked[1] :

- **Will the user try to achieve the effect that the subtask has?** Does the user understand that this subtask is needed to reach the user's goal?
- **Will the user notice that the correct action is available?** E.g. is the button visible?
- **Will the user understand that the wanted subtask can be achieved by the action?** E.g. the right button is visible but the user does not understand the text and will therefore not click on it.
- **Does the user get feedback?** Will the user know that they have done the right thing after performing the action?

By answering the questions for each subtask usability problems will be noticed.

Common Mistakes

In teaching people to use the walkthrough method, Lewis & Rieman have found that there are two common misunderstandings [2] :

1. The evaluator doesn't know how to perform the task themself, so they stumble through the interface trying to discover the correct sequence of actions -- and then they evaluate the stumbling process. (The user should identify and perform the **optimal** action sequence.)
2. The walkthrough does not test real users on the system. The walkthrough will often identify many more problems than you would find with a single, unique user in a single test session.

History

The method was developed in the early nineties by Wharton, et al., and reached a large usability audience when it was published as a chapter in Jakob Nielsen's seminal book on usability, "Usability Inspection Methods." The Wharton, et al. method required asking four questions at each step, along with extensive documentation of the analysis. In 2000 there was a resurgence in interest in the method in response to a CHI paper by Spencer who described modifications to the method to make it effective in a real software development setting. Spencer's streamlined method required asking only two questions at each step, and involved creating less documentation. Spencer's paper followed the example set by Rowley, et al. who described the modifications to the method that they made based on their experience applying the methods in their 1992 CHI paper "The Cognitive Jogthrough".

References

[1] C. Wharton et al. "The cognitive walkthrough method: a practitioner's guide" in J. Nielsen & R. Mack "Usability Inspection Methods" pp. 105-140.
[2] http://hcibib.org/tcuid/chap-4.html#4-1

Further reading

- Blackmon, M. H. Polson, P.G. Muneo, K & Lewis, C. (2002) Cognitive Walkthrough for the Web CHI 2002 vol.4 No.1 pp463–470
- Blackmon, M. H. Polson, Kitajima, M. (2003) Repairing Usability Problems Identified by the Cognitive Walkthrough for the Web CHI 2003 pp497–504.
- Dix, A., Finlay, J., Abowd, G., D., & Beale, R. (2004). Human-computer interaction (3rd ed.). Harlow, England: Pearson Education Limited. p321.
- Gabrielli, S. Mirabella, V. Kimani, S. Catarci, T. (2005) Supporting Cognitive Walkthrough with Video Data: A Mobile Learning Evaluation Study MobileHCI ’05 pp77–82.

- Goillau, P., Woodward, V., Kelly, C. & Banks, G. (1998) Evaluation of virtual prototypes for air traffic control - the MACAW technique. In, M. Hanson (Ed.) Contemporary Ergonomics 1998.
- Good, N. S. & Krekelberg, A. (2003) Usability and Privacy: a study of KaZaA P2P file-sharing CHI 2003 Vol.5 no.1 pp137–144.
- Gray, W. & Salzman, M. (1998). Damaged merchandise? A review of experiments that compare usability evaluation methods, *Human-Computer Interaction vol.13 no.3*, 203-61.
- Gray, W.D. & Salzman, M.C. (1998) Repairing Damaged Merchandise: A rejoinder. Human-Computer Interaction vol.13 no.3 pp325–335.
- Hornbaek, K. & Frokjaer, E. (2005) Comparing Usability Problems and Redesign Proposal as Input to Practical Systems Development CHI 2005 391-400.
- Jeffries, R. Miller, J. R. Wharton, C. Uyeda, K. M. (1991) User Interface Evaluation in the Real World: A comparison of Four Techniques Conference on Human Factors in Computing Systems pp 119 – 124
- Lewis, C. Polson, P, Wharton, C. & Rieman, J. (1990) Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces Chi ’90 Proceedings pp235–242.
- Mahatody, Thomas / Sagar, Mouldi / Kolski, Christophe (2010). State of the Art on the Cognitive Walkthrough Method, Its Variants and Evolutions, International Journal of Human-Computer Interaction, 2, 8 741-785.
- Rowley, David E., and Rhoades, David G (1992). The Cognitive Jogthrough: A Fast-Paced User Interface Evaluation Procedure. *Proceedings of CHI '92*, 389-395.
- Sears, A. (1998) The Effect of Task Description Detail on Evaluator Performance with Cognitive Walkthroughs CHI 1998 pp259–260.
- Spencer, R. (2000) The Streamlined Cognitive Walkthrough Method, Working Around Social Constraints Encountered in a Software Development Company CHI 2000 vol.2 issue 1 pp353–359.
- Wharton, C. Bradford, J. Jeffries, J. Franzke, M. Applying Cognitive Walkthroughs to more Complex User Interfaces: Experiences, Issues and Recommendations CHI ’92 pp381–388.

External links

- Cognitive Walkthrough (http://www.pages.drexel.edu/~zwz22/CognWalk.htm)

Heuristic evaluation

A **heuristic evaluation** is a discount usability inspection method for computer software that helps to identify usability problems in the user interface (UI) design. It specifically involves evaluators examining the interface and judging its compliance with recognized usability principles (the "heuristics"). These evaluation methods are now widely taught and practiced in the New Media sector, where UIs are often designed in a short space of time on a budget that may restrict the amount of money available to provide for other types of interface testing.

Introduction

The main goal of heuristic evaluations is to identify any problems associated with the design of user interfaces. Usability consultant Jakob Nielsen developed this method on the basis of several years of experience in teaching and consulting about usability engineering.

Heuristic evaluations are one of the most informal methods[1] of usability inspection in the field of human-computer interaction. There are many sets of usability design heuristics; they are not mutually exclusive and cover many of the same aspects of user interface design.

Quite often, usability problems that are discovered are categorized—often on a numeric scale—according to their estimated impact on user performance or acceptance. Often the heuristic evaluation is conducted in the context of use cases (typical user tasks), to provide feedback to the developers on the extent to which the interface is likely to be compatible with the intended users' needs and preferences.

The simplicity of heuristic evaluation is beneficial at the early stages of design. This usability inspection method does not require user testing which can be burdensome due to the need for users, a place to test them and a payment for their time. Heuristic evaluation requires only one expert, reducing the complexity and expended time for evaluation. Most heuristic evaluations can be accomplished in a matter of days. The time required varies with the size of the artifact, its complexity, the purpose of the review, the nature of the usability issues that arise in the review, and the competence of the reviewers. Using heuristic evaluation prior to user testing will reduce the number and severity of design errors discovered by users. Although heuristic evaluation can uncover many major usability issues in a short period of time, a criticism that is often leveled is that results are highly influenced by the knowledge of the expert reviewer(s). This "one-sided" review repeatedly has different results than performance testing, each type of testing uncovering a different set of problems.

Nielsen's heuristics

Jakob Nielsen's heuristics are probably the most-used usability heuristics for user interface design. Nielsen developed the heuristics based on work together with Rolf Molich in 1990.[1] [2] The final set of heuristics that are still used today were released by Nielsen in 1994.[3] The heuristics as published in Nielsen's book *Usability Engineering* are as follows[4]

Visibility of system status:
The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

Match between system and the real world:
The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom:
Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards:
Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention:
Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Recognition rather than recall:
Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use:
Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Aesthetic and minimalist design:
Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Help users recognize, diagnose, and recover from errors:
Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation:
Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Gerhardt-Powals’ cognitive engineering principles

Although Nielsen is considered the expert and field leader in heuristics, Jill Gerhardt-Powals [5] also developed a set of cognitive principles for enhancing computer performance.[6] These heuristics, or principles, are similar to Nielsen’s heuristics but take a more holistic approach to evaluation. Gerhardt Powals’ principles[7] are listed below.

- **Automate unwanted workload**:
  - free cognitive resources for high-level tasks.
  - eliminate mental calculations, estimations, comparisons, and unnecessary thinking.
- **Reduce uncertainty**:
  - display data in a manner that is clear and obvious.
- **Fuse data**:
  - reduce cognitive load by bringing together lower level data into a higher-level summation.
- **Present new information with meaningful aids to interpretation**:
  - use a familiar framework, making it easier to absorb.
  - use everyday terms, metaphors, etc.
- **Use names that are conceptually related to function**:
  - Context-dependent.
  - Attempt to improve recall and recognition.
  - Group data in consistently meaningful ways to decrease search time.

- **Limit data-driven tasks**:
  - Reduce the time spent assimilating raw data.
  - Make appropriate use of color and graphics.
- **Include in the displays only that information needed by the user at a given time**.
- **Provide multiple coding of data when appropriate**.
- **Practice judicious redundancy**.

References

[1] Nielsen, J., and Molich, R. (1990). Heuristic evaluation of user interfaces, Proc. ACM CHI'90 Conf. (Seattle, WA, 1–5 April), 249-256
[2] Molich, R., and Nielsen, J. (1990). Improving a human-computer dialogue, Communications of the ACM 33, 3 (March), 338-348
[3] Nielsen, J. (1994). Heuristic evaluation. In Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, John Wiley & Sons, New York, NY
[4] Nielsen, Jakob (1994). *Usability Engineering*. San Diego: Academic Press. pp. 115–148. ISBN 0-12-518406-9.
[5] http://loki.stockton.edu/~gerhardj/
[6] [|Gerhardt-Powals, Jill (http://loki.stockton.edu/~gerhardj/)] (1996). "Cognitive engineering principles for enhancing human - computer performance". *International Journal of Human-Computer Interaction* **8** (2): 189–211.
[7] Heuristic Evaluation - Usability Methods – What is a heuristic evaluation? (http://usability.gov/methods/test_refine/heuristic.html#WhatisaHeuristicEvaluation) Usability.gov

External links

- Jakob Nielsen's introduction to Heuristic Evaluation (http://www.useit.com/papers/heuristic/) - Including fundamental points, methodologies and benefits.
- Alternate First Principles (Tognazzini) (http://www.asktog.com/basics/firstPrinciples.html) - Including Jakob Nielsen's ten rules of thumb
- Heuristic Evaluation at Usability.gov (http://www.usability.gov/methods/heuristiceval.html)
- Heuristic Evaluation in the RKBExplorer (http://www.rkbexplorer.com/explorer/#display=mechanism-{http://resex.rkbexplorer.com/id/resilience-mechanism-4331d919})

Further reading

- Dix, A., Finlay, J., Abowd, G., D., & Beale, R. (2004). Human-computer interaction (3rd ed.). Harlow, England: Pearson Education Limited. p324
- Gerhardt-Powals, Jill (1996). Cognitive Engineering Principles for Enhancing Human-Computer Performance. “International Journal of Human-Computer Interaction”, 8(2), 189-21
- Hvannberg, E., Law, E., & Lárusdóttir, M. (2007) “Heuristic Evaluation: Comparing Ways of Finding and Reporting Usability Problems”, Interacting with Computers, 19 (2), 225-240
- Nielsen, J. and Mack, R.L. (Eds) (1994). Usability Inspection Methods, John Wiley & Sons Inc

Pluralistic walkthrough

The **Pluralistic Walkthrough** (also called a **Participatory Design Review**, **User-Centered Walkthrough**, **Storyboarding**, **Table-Topping**, or **Group Walkthrough**) is a usability inspection method used to identify usability issues in a piece of software or website in an effort to create a maximally usable human-computer interface. The method centers around using a group of users, developers and usability professionals to step through a task scenario, discussing usability issues associated with dialog elements involved in the scenario steps. The group of experts used is asked to assume the role of typical users in the testing. The method is prized for its ability to be utilized at the earliest design stages, enabling the resolution of usability issues quickly and early in the design process. The method also allows for the detection of a greater number of usability problems to be found at one time due to the interaction of multiple types of participants (users, developers and usability professionals). This type of usability inspection method has the additional objective of increasing developers' sensitivity to users' concerns about the product design.

Procedure

Walkthrough Team

A walkthrough team must be assembled prior to the pluralistic walkthrough. Three types of participants are included in the walkthrough: representative users, product developers and human factors (usability) engineers/professionals. Users should be representative of the target audience, and are considered the primary participants in the usability evaluation. Product developers answer questions about design and suggest solutions to interface problems users have encountered. Human factors professionals usually serve as the facilitators and are also there to provide feedback on the design as well as recommend design improvements. The role of the facilitator is to guide users through tasks and facilitate collaboration between users and developers. It is best to avoid having a product developer assume the role of facilitator, as they can get defensive to criticism of their product.

Materials

The following materials are needed to conduct a pluralistic walkthrough:

- Room large enough to accommodate approximately 6-10 users, 6-10 developers and 2-3 usability engineers
- Printed screen-shots (paper prototypes) put together in packets in the same order that the screens would be displayed when users were carrying out the specific tasks. This includes hard copy panels of screens, dialog boxes, menus, etc presented in order.
- Hard copy of the task scenario for each participant. There are several scenarios defined in this document complete with the data to be manipulated for the task. Each participant receives a package that enables him or her to write a response (i.e. the action to take on that panel) directly onto the page. The task descriptions for the participant are short direct statements.
- Writing utensils for marking up screen shots and filling out documentation and questionnaires.

Participants are given written instructions and rules at the beginning of the walkthrough session. The rules indicate to all participants (users, designers, usability engineers) to:

- Assume the role of the user
- To write on the panels the actions they would take in pursuing the task at hand
- To write any additional comments about the task
- Not flip ahead to other panels until they are told to
- To hold discussion on each panel until the facilitator decides to move on

Tasks

Pluralistic walkthroughs are group activities that require the following steps be followed:

1. Participants are presented with the instructions and the ground rules mentioned above. The task description and scenario package are also distributed.
2. Next, a product expert (usually a product developer) gives a brief overview of key product concepts and interface features. This overview serves the purpose of stimulating the participants to envision the ultimate final product (software or website), so that the participants would gain the same knowledge and expectations of the ultimate product that product end users are assumed to have.
3. The usability testing then begins. The scenarios are presented to the panel of participants and they are asked to write down the sequence of actions they would take in attempting to complete the specified task (i.e. moving from one screen to another). They do this individually without conferring amongst each other.
4. Once everyone has written down their actions independently, the participants discuss the actions that they suggested for that task. They also discuss potential usability problems. The order of communication is usually such that the representative users go first so that they are not influenced by the other panel members and are not deterred from speaking.
5. After the users have finished, the usability experts present their findings to the group. The developers often explain their rationale behind their design. It is imperative that the developers assume an attitude of welcoming comments that are intended to improve the usability of their product.
6. The walkthrough facilitator presents the correct answer if the discussion is off course and clarifies any unclear situations.
7. After each task, the participants are given a brief questionnaire regarding the usability of the interface they have just evaluated.
8. Then the panel moves on to the next task and round of screens. This process continues until all the scenarios have been evaluated.

Throughout this process, usability problems are identified and classified for future action. The presence of the various types of participants in the group allows for a potential synergy to develop that often leads to creative and collaborative solutions. This allows for a focus on user-centered perspective while also considering the engineering constraints of practical system design.

Characteristics of Pluralistic Walkthrough

Other types of usability inspection methods include: Cognitive Walkthroughs, Interviews, Focus Groups, Remote Testing and Think Aloud Protocol. Pluralistic Walkthroughs share some of the same characteristics with these other traditional walkthroughs, especially with cognitive walkthroughs, but there are some defining characteristics (Nielsen, 1994):

- The main modification, with respect to usability walkthroughs, was to include three types of participants: representative users, product developers, and human factors (usability) professionals.
- Hard-copy screens (panels) are presented in the same order in which they would appear online. A task scenario is defined, and participants confront the screens in a linear path, through a series of user interface panels, just as they would during the successful conduct of the specified task online, as the site/software is currently designed.
- Participants are all asked to assume the role of the user for whatever user population is being tested. Thus, the developers and the usability professionals are supposed to try to put themselves in the place of the users when making written responses.
- The participants write down the action they would take in pursuing the designated task online, before any further discussion is made. Participants are asked to write their responses in as much detail as possible down to the keystroke or other input action level. These written responses allow for some production of quantitative data on user actions that can be of value.

- It is only after all participants have written the actions they would take that discussion would begin. The representative users offer their discussion first and discuss each scenario step. Only after the users have exhausted their suggestions do the usability experts and product developers offer their opinions.

Benefits and Limitations

Benefits

There are several benefits that make the pluralistic usability walkthrough a valuable tool.

- Early systematic look at a new product, gaining early performance and satisfaction data from users about a product. Can provide early performance and satisfaction data before costly design strategies have been implemented.
- Strong focus on user centered design in task analysis, leading to more problems identified at an earlier point in development. This reduces the iterative test-redesign cycle by utilizing immediate feedback and discussion of design problems and possible solutions while users are present.
- Synergistic redesign because of the group process involving users, developers and usability engineers. The discussion of the identified problems in a multidisciplinary team will spawn creative, usable and quick solutions.
- Valuable quantitative and qualitative data is generated through users' actions documented by written responses.
- Product developers at the session gain appreciation for common user problems, frustrations or concerns regarding the product design. Developers become more sensitive to users' concerns.

Limitations

There are several limitations to the pluralistic usability walkthrough that affect its usage.

- The walkthrough can only progress as quickly as the slowest person on each panel. The walkthrough is a group exercise and, therefore, in order to discuss a task/screen as a group, we must wait for all participants to have written down their responses to the scenario. The session can feel laborious if too slow.
- A fairly large group of users, developers and usability experts has to be assembled at the same time. Scheduling could be a problem.
- All the possible actions can't be simulated on hard copy. Only one viable path of interest is selected per scenario. This precludes participants from browsing and exploring, behaviors that often lead to additional learning about the user interface.
- Product developers might not feel comfortable hearing criticism about their designs.
- Only a limited number of scenarios (i.e. paths through the interface) can be explored due to time constraints.
- Only a limited amount of recommendations can be discussed due to time constraints.

Further reading

- Dix, A., Finlay, J., Abowd, G., D., and Beale, R. Human-computer interaction (3rd ed.). Harlow, England: Pearson Education Limited, 2004.
- Nielsen, Jakob. Usability Inspection Methods. New York, NY: John Wiley and Sons, 1994.
- Preece, J., Rogers, Y., and Sharp, H. Interaction Design. New York, NY: John Wiley and Sons, 2002.
- Bias, Randolph G., "The Pluralistic Usability Walkthrough: Coordinated Emphathies," in Nielsen, Jakob, and Mack, R. eds, Usability Inspection Methods. New York, NY: John Wiley and Sons. 1994.

External links

- List of Usability Evaluation Methods and Techniques [1]
- Pluralistic Usability Walkthrough [2]

References

[1] http://www.usabilityhome.com/FramedLi.htm?PlurWalk.htm
[2] http://www.usabilitybok.org/methods/p2049

Comparison of usability evaluation methods

Evaluation Method	Evaluation Method Type	Applicable Stages	Description	Advantages	Disadvantages
Think aloud protocol	Testing	Design, coding, testing and release of application	Participants in testing express their thoughts on the application while executing set tasks	• Less expensive • Results are close to what is experienced by users	• The Environment is not natural to the user
Remote testing	Testing	Design, coding, testing and release of application	The experimenter does not directly observe the users while they use the application	• Efficiency, effectiveness and satisfaction, the three usability issues, are covered	• Additional Software is necessary to observe the participants from a distance
Focus groups	Inquiry	Testing and release of application	A moderator guides a discussion with a group of users of the application	• If done before prototypes are developed, can save money • Produces a lot of useful ideas from the users themselves • Can improve customer relations	• The environment is not natural to the user and may provide inaccurate results. • The data collected tends to have low validity due to the unstructured nature of the discussion
Interviews	Inquiry	Design, coding, testing and release of application	The users are interviewed to find out about their experience and expectations	• Good at obtaining detailed information • Few participants are needed • Can improve customer relations	• Can not be conducted remotely • Does not address the usability issue of efficiency
Cognitive walkthrough	Inspection	Design, coding, testing and release of application	A team of evaluators walk through the application discussing usability issues through the use of a paper prototype or a working prototype	• Good at refining requirements • does not require a fully functional prototype	• Does not address user satisfaction or efficiency • The designer may not behave as the average user when using the application
Pluralistic walkthrough	Inspection	Design	A team of users, usability engineers and product developers review the usability of the paper prototype of the application	• Usability issues are resolved faster • Greater number of usability problems can be found at one time	• Does not address the usability issue of efficiency

Source: Genise, Pauline. “Usability Evaluation: Methods and Techniques: Version 2.0” August 28, 2002. University of Texas.

Article Sources and Contributors

Software testing *Source*: http://en.wikipedia.org/w/index.php?oldid=446402745 *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, Ahy1, Aitias, Akamad, Akhiladi007, AlMac, Alappuzhakaran, Albanaco, Alhenry2006, AliaksandrAA, AliveFreeHappy, Allan McInnes, Allstarecho, Alvestrand, Amire80, Amty4all, Andonic, Andre Engels, Andreas Kaufmann, Andres, Andrewcmcardle, 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, Bigwyrm, Bilbo1507, Bindu Laxminarayan, Bkil, 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, ChiLlBeserker, 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, Dacoutts, 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, G0gogcsc300, GABaker, Gail, Gar3t, Gary King, Gary Kirk, Gdavidp, Gdo01, GeoTe, Georgie Canadian, Geosak, Giggy, Gil mo, Gogo Dodo, Goldom, Gonchibolso12, Gorson78, GraemeL, GregorB, Gsmgm, Guehene, Gurchzilla, GururajOaksys, Guybrush1979, Hadal, Halovivek, Halsteadk, HamburgerRadio, Harald Hansen, Havlatm, Haza-w, Hdt83, Headbomb, Helix84, Hemnath18, Henri662, Honey88foru, Hooperbloob, Hsingh77, Hu12, Hubschrauber729, Huge Bananas, Hutch1989r15, I dream of horses, IJA, IceManBrazil, Ignasiokambale, ImALion, Imroy, Incnis Mrsi, Indon, Infrogmation, Intray, Inwind, J.delanoy, JASpencer, JPFitzmaurice, Ja 62, JacobBramley, Jake Wartenberg, Jeff G., Jehochman, Jenny MacKinnon, JesseHogan, JimD, Jjamison, Jluedem, Jm266, Jmax-, Jmckey, JoeSmack, John S Eden, Johndci, Johnny.cache, Johnuniq, JonJosephA, Joneskoo, JosephDonahue, Josheisenberg, Joshymit, Joyous!, Jsled, Jstastny, Jtowler, Juliancolton, JuneGloom07, Jwoodger, 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, Mhaitham.shammaa, Michael B. Trausch, Michael Bernstein, MichaelBolton, Michig, Mike Doughney, MikeDogma, Miker@sundialservices.com, Mikethegreen, Misza13, Mitch Ames, Miterdale, Mmgreiner, Moa3333, Mpilaeten, 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, Notinasnaid, Novice7, Nuno Tavares, Oashi, Ocee, Oddity-, Ohnoitsjamie, Oicumayberight, Oliver1234, Omicronpersei8, Orange Suede Sofa, Orphan Wiki, Ospalh, Otis80hobson, Oysterguitarist, PL290, Paranomia, Pascal.Tesson, Pashute, Paudelp, Paul August, Paul.h, Pcb21, Peashy, Pepsi12, PhilHibbs, Philip Trueman, PhilipO, PhilippeAntras, Phoe6, Piano non troppo, Pieleric, Pinecar, Plainplow, Pmberry, Pointillist, Pomoxis, Poulpy, Pplolpp, Prari, Praveen.karri, Priya4212, Promoa1, Psychade, Puraniksameer, Pysuresh, QTCaptain, Qaiassist, Qatutor, Qazwsxedcrfvtgbyhn, Qwyrxian, RHaworth, Radagast83, Rahuljaitley82, Rajesh mathur, RameshaLB, Randhirreddy, Ravialluru, Raynald, RedWolf, RekishiEJ, Remi0o, ReneS, Retired username, Rex black, Rgoodermote, Rhobite, Riagu, Rich Farmbrough, Richard Harvey, RitigalaJayasena, Rje, Rjwilmsi, Rlsheehan, Rmattson, Rmstein, Robbie098, Robert Merkel, Robinson weijman, Rockynook, Ronhjones, Ronz, Roscelese, Rowlye, Rp, Rror, Ruptan, Rwwww, S.K., SJP, SP-KP, SURIV, Sachipra, Sachxn, Sam Hocevar, Samansouri, Sankshah, Sapphic, Sardanaphalus, Sasquatch525, SatishKumarB, ScaledLizard, ScottSteiner, Scottri, Sega381, Selket, Senatum, Serge Toper, Sergeyl1984, Shadowcheets, Shanes, Shepmaster, Shimeru, Shimgray, Shishirhegde, Shoejar, Shubo mu, Shze, Silverbullet234, Sitush, Skalra7, Skyqa, Slowbro, Smack, Smurrayinchester, Snowolf, Softtest123, Softwaretest1, Softwaretesting1001, Softwaretesting101, Solde, Somdeb Chakraborty, Someguy1221, Sooner Dave, SpaceFlight89, Spadoink, SpigotMap, Spitfire, Srikant.sharma, Staceyeschneider, Stansult, StaticGull, Stephen Gilbert, Stephenb, Steveozone, Stickee, Storm Rider, Strmore, SunSw0rd, Superbeecat, SwirlBoy39, Sxm20, Sylvainmarquis, T4tarzan, TCL India, Tagro82, Tdjones74021, Techsmith, Tedickey, Tejas81, Terrillja, 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, Tprosser, Ttam, Tulkolahten, Tusharpandya, TutterMouse, Uktim63, Uncle G, Unforgettableid, Useight, Utcursch, Uzma Gamal, VMS Mosaic, Valenciano, Vaniac, Venkatreddyc, Venu6132000, Verloren, VernoWhitney, Versageek, Vijay.ram.pm, Vijaythormothe, Vishwas008, Vsoid, W2qasource, Walter Görlitz, Wifione, Wiki alf, WikiWilliamP, Wikieditor06, Will Beback Auto, Willsmith, Winchelsea, Wlievens, Wombat77, Wwmbes, Yamamoto Ichiro, Yesyoubee, Yngupta, Yosri, Yuckfoo, ZenerV, Zephyrjs, ZhonghuaDragon2, ZooFari, Zurishaddai, 1992 anonymous edits

Black-box testing *Source*: http://en.wikipedia.org/w/index.php?oldid=442898546 *Contributors*: A bit iffy, AKGhetto, Aervanath, Ag2402, AndreniW, Andrewpmk, Ash, Asparagus, Benito78, Betterusername, Blake-, CWY2190, Caesura, Canterbury Tail, Chris Pickett, Chrys, ClementSeveillac, Colinky, Courcelles, DRogers, DanDoughty, Daveydweeb, Discospinster, DividedByNegativeZero, Docboat, DylanW, Ebde, Electiontechnology, Epim, Erkan Yilmaz, ErkinBatu, Fluzwup, Frap, Gayathri nambiar, Geeoharee, Haymaker, Hooperbloob, Hu12, Hugh.glaser, Ian Pitchford, Ileshko, Isnow, Jmabel, Jondel, Karl Naylor, Kgf0, Khym Chanur, Kuru, LOL, Lahiru k, Lambchop, Liao, Mark.murphy, Mathieu, Michael Hardy, Michig, Mpilaeten, Mr Minchin, MrOllie, NEUrOO, NawlinWiki, Nitinqai, Notinasnaid, OlEnglish, Otheus, PAS, PerformanceTester, Picaroon, Pinecar, Poor Yorick, Pradameinhoff, Radiojon, Retiono Virginian, Rich Farmbrough, Rstens, Rsutherland, Rwwww, S.K., Sergei, Shadowjams, Shijaz, Solde, Subversive.sound, SuperMidget, Tedickey, Tobias Bergemann, Toddst1, UnitedStatesian, WJBscribe, Walter Görlitz, Xaosflux, Zephyrjs, 207 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

San Francisco depot *Source*: http://en.wikipedia.org/w/index.php?oldid=362281144 *Contributors*: Andreas Kaufmann, Auntof6, Centrx, DRogers, EagleFan, Fabrictramp, PigFlu Oink, Pinecar

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

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

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, Wisgary, Zoz, 34 anonymous edits

Boundary-value analysis *Source*: http://en.wikipedia.org/w/index.php?oldid=440330386 *Contributors*: Ahoerstemeier, Andreas Kaufmann, AndreniW, Attilios, Benito78, Ccady, DRogers, Duggpm, Ebde, Eumolpo, Freek Verkerk, Ianr44, IceManBrazil, Jtowler, Krishjugal, Linuxbabu, Michaeldunn123, Mirokado, Pinecar, Psiphiorg, Radiojon, Retired username, Robinson weijman, Ruchir1102, Sesh, Stemburn, Stemonitis, Sunithasiri, Velella, Walter Görlitz, Wisgary, Zoz, 59 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

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, Dirkbb, Doradus, Edward, Emurphy42, Enric 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, Marqueed, Martinmeyer, Marudubshinki, McGeddon, Mezzaluna, MikeEddington, Monty845, Mpeisenbr, MrOllie, Nandhp, Neale Monks, Neelix, Niri.M, Pinecar, Posix memalign, Povman, Rcsprinter123, Ronz, Sadeq, Softtest123, Starofale, Stephanakib, Stevehughes, SwissPokey, T0pgear09, The Anome, The Cunctator, Tmaufer, Tremilux, User At Work, Victor Stinner, Walter Görlitz, Yurymik, Zarkthehackeralliance, Zippy, Zirconscot, 138 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, Wleizero, 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, Jtowler, Jzander, Kku, MDE, Mark Renier, MarkUtting, Mattisse, Mdd, Michael Hardy, Micskeiz, Mirko.conrad, Mjchonoles, MrOllie, Pinecar, Richard R White, S.K., Sdorrance, Smartesting, Suka, Tatzelworm, Tedickey, Test-tools, That Guy, From That Show!, TheParanoidOne, Thv, Vonkje, Williamglasby, Yan Kuligin, Yxl01, 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, Jetfreeman, Jwoodger, KarlDubost, MER-C, Macrofiend, Narayanraman, P199, Pinecar, Rchandra, Runnerweb, SEWilco, Softtest123, Testgeek, Thadius856, TubularWorld,

Walter Görlitz, 32 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

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, Err0neous, 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, Velella, Walter Görlitz, Yadyn, Yilloslime, 114 anonymous edits

Code coverage *Source*: http://en.wikipedia.org/w/index.php?oldid=446190361 *Contributors*: 194.237.150.xxx, Abdull, Abednigo, Ad88110, Agasta, Aislingdonnelly, 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, JorisvS, Jtheires, Julias.shaw, Kdakin, Kku, Kurykh, LDRA, LouScheffer, M4gnum0n, MER-C, Materialscientist, Mati22081979, Matt Crypto, Millerlyte87, Miracleworker5263, Mj1000, MrOllie, MywikiaccountSA, Nat hillary, NawlinWiki, Nigelj, Nin1975, Nixeagle, Ntalamai, Parasoft-pl, Penumbra2000, Phatom87, Pinecar, Ptrb, Quamrana, Quinntaylor, Quux, RedWolf, Roadbiker53, Robert Merkel, Rpapo, RuggeroB, Rwwww, Scubamunki, Sebastian.Dietrich, SimonKagstrom, Smharr4, Snoyes, Suruena, Taibah U, Technoparkcorp, Test-tools, Testcocoon, Tiagofassoni, TutterMouse, U2perkunas, Veralift, Walter Görlitz, WimdeValk, Witten rules, Wlievens, Wmwmurray, X746e, 217 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

Fault injection *Source*: http://en.wikipedia.org/w/index.php?oldid=446109779 *Contributors*: Andreas Kaufmann, Ari.takanen, Auntof6, CapitalR, Chowbok, CyborgTosser, DaGizza, DatabACE, Firealwaysworks, Foobiker, Jeff G., Joriki, Paff1, Paul.Dan.Marinescu, Piano non troppo, RHaworth, SteveLoughran, Suruena, Tmaufer, Tony1, WillDo, 26 anonymous edits

Bebugging *Source*: http://en.wikipedia.org/w/index.php?oldid=409176429 *Contributors*: Andreas Kaufmann, Dawynn, Erkan Yilmaz, Foobiker, Jchaw, Kaihsu, O keyes, 6 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, Felixwikihudson, Fuhghettaboutit, GiuseppeDiGuglielmo, Htmlapps, Jarfil, Jeffoffutt, JonHarder, Martpol, Mycroft.Holmes, Pieleric, Pinecar, Quuxplusone, Rohansahgal, Sae1962, Usrnme h8er, Walter Görlitz, Wikid77, Yuejia, 47 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

Software performance testing *Source*: http://en.wikipedia.org/w/index.php?oldid=444211955 *Contributors*: AMbroodEY, AbsolutDan, Alex Vinokur, Andreas Kaufmann, Apodelko, Argyriou, Armadillo-eleven, Bbryson, Bourgeoisspy, Brian.a.wilson, Burakseren, Cit helper, Ckoenigsberg, Coroberti, D6, David Johnson, Davidschmelzer, Deicool, Dhiraj1984, Dwvisser, 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, Muhandes, Mywikicontribs, Nono64, Notinasnaid, Oliver Lineham, Optakeover, Pinecar, Pratheepraj, Ravialluru, Raysecurity, Rjwilmsi, Robert Merkel, Ronz, Rsbarber, Rstens, Sebastian.Dietrich, ShelfSkewed, Shimser, Shirtwaist, Shoeofdeath, SimonP, Softlogica, SunSw0rd, Swtechwr, Timgurto, Veinor, Versageek, Wahab80, Walter Görlitz, Weregerbil, Wilsonmar, Wizzard, Wktsugue, Woohookitty, Wselph, 235 anonymous edits

Stress testing *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

Load testing *Source*: http://en.wikipedia.org/w/index.php?oldid=443731698 *Contributors*: AbsolutDan, ArrowmanCoder, BD2412, Bbryson, Belmond, Bernard2, CanadianLinuxUser, Crossdader, Czei, Derby-ridgeback, Dhiraj1984, El Tonerino, Ettrig, Faught, 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.maclean, 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

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

Scalability testing *Source*: http://en.wikipedia.org/w/index.php?oldid=411550068 *Contributors*: GregorB, JaGa, Kumar74, Malcolma, Methylgrace, Pinecar, 6 anonymous edits

Compatibility testing *Source*: http://en.wikipedia.org/w/index.php?oldid=370597747 *Contributors*: Alison9, Arkitus, Iain99, Jimj wpg, Kumar74, Neelov, Pinecar, RekishiEJ, Rwwww, 4 anonymous edits

Portability testing *Source*: http://en.wikipedia.org/w/index.php?oldid=415815752 *Contributors*: Andreas Kaufmann, Biscuittin, Cmdrjameson, Nibblus, OSborn, Pharos, Tapir Terrific, The Public Voice, 2 anonymous edits

Security testing *Source*: http://en.wikipedia.org/w/index.php?oldid=445168305 *Contributors*: Aaravind, Andreas Kaufmann, Bigtimepeace, Bwpach, ConCompS, DanielPharos, David Stubley, Dxwell, Epbr123, Gardener60, Gavenko a, Glane23, ImperatorExercitus, JonHarder, Joneskoo, Kinu, Lotje, MichaelBillington, Pinecar, Ravi.alluru@applabs.com, Shadowjams, Softwaretest1, Someguy1221, Spitfire, Stenaught, ThisIsAce, Uncle Milty, WereSpielChequers, 98 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, Smokizzy, 3 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

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, Nlhenk, Pinecar, Pnm, Thumperward, Traveler78, Vipinhari, 10 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

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

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, Jjamison, Jkl, Mathiastck, PhilippeAntras, Pinecar, Robofish, Swtechwr, Ulner, 12 anonymous edits

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, Anorthup, Ardonik, Asavoia, Attilios, Autarch, Bakersg13, Bdijkstra, BenFrantzDale, Brian Geppert, CanisRufus, Canterbury Tail, Chris Pickett, ChristianEdwardGruber, ChuckEsterbrook, Clausen, Colonies Chris, Corvi, Craigwb, DRogers, DanMS, Derbeth, Dillard421, Discospinster, Dmulter, Earlypsychosis, Edaelon, Edward Z. Yang, Eewild, El T, Elilo, Evil saltine, Excirial, FlashSheridan, FrankTobia, Fredrik, Furrykef, GTBacchus, Gggggdxn, Goswamivijay, Guille.hoardings, 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, Mcsee, Mheusser, Mhhanley, Michig, MickeyWiki, Miker@sundialservices.com, Mild Bill Hiccup, Mortense, Mr. Disguise, MrOllie, Mtomczak, Nat hillary, Nate Silva, Nbryant, Neilc, Nick Lewis CNH, Notinasnaid, Ohnoitsjamie, OmriSegal, Ottawa4ever, PGWG, Pablasso, Paling Alchemist, Pantosys, Paul August, Paulocheque, Pcb21, 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, Sligocki, Smalljim, So God created Manchester, Solde, Sozin, Ssd, Sspiro, Stephenb, SteveLoughran, Stumps, Sujith.srao, Svick, Swtechwr, Sybersnake, TFriesen, Themillofkeytone, Thv, Timo Honkasalo, Tlroche, Tobias Bergemann, Toddst1, Tony Morris, Tyler Oderkirk, Unittester123, User77764, VMS Mosaic, Veghead, Vishnava, Walter Görlitz, Willem-Paul, Winhunter, Wmahan, Zed toocool, 450 anonymous edits

Self-testing code *Source*: http://en.wikipedia.org/w/index.php?oldid=326302253 *Contributors*: Andreas Kaufmann, Ed Poor, GregorB, Malcolma, Rich Farmbrough, Spoon!, 2 anonymous edits

Test fixture *Source*: http://en.wikipedia.org/w/index.php?oldid=438605439 *Contributors*: Andreas Kaufmann, Brambleclawx, Heathd, Humanoc, Ingeniero-aleman, Jeodesic, Martarius, Pkgx, RCHenningsgard, Ripounet, Rlsheehan, Silencer1981, Tabletop, WHonekamp, Walter Görlitz, Wernight, ZacParkplatz, 16 anonymous edits

Method stub *Source*: http://en.wikipedia.org/w/index.php?oldid=423993207 *Contributors*: Andreas Kaufmann, Antonielly, Bhadani, Bratch, Can't sleep, clown will eat me, Cander0000, Ceyockey, Dasoman, Deep Alexander, Dicklyon, Drbreznjev, Ermey, Extvia, Ggoddard, Hollih, IguanaScales, Itai, Joaopaulo1511, Kku, MBisanz, Mange01, Mark Renier, Michig, Mityaha, Pinecar, Radagast83, Rich Farmbrough, RitigalaJayasena, Rrburke, S.K., Segv11, Sj, Thisarticleisastub, Vary, Walter Görlitz, 31 anonymous edits

Mock object *Source*: http://en.wikipedia.org/w/index.php?oldid=446411485 *Contributors*: 16x9, A. B., ABF, AN(Ger), Acather96, Allanlewis, Allen Moore, Andreas Kaufmann, Andy Dingley, Antonielly, Ataru, Autarch, Babomb, BenWilliamson, Blueboy96, Charles Matthews, Ciphers, ClinkingDog, CodeCaster, Colcas, Cst17, Cybercobra, DHGarrette, Dcamp314, Derbeth, Dhoerl, Edward Z. Yang, Elilo, Ellissound, Eric Le Bigot, Ghettoblaster, Hooperbloob, IceManBrazil, JamesShore, Kc8tpz, Khalid hassani, Kku, Le-sens-commun, Lmajano, Lotje, Mange01, Marchaos, Martarius, Martinig, MaxSem, Mkarlesky, NickHodges, Nigelj, Nrabinowitz, Paul Foxworthy, Pecaperopeli, Philip Trueman, Pinecar, R'n'B, Redeagle688, Rodrigez, RoyOsherove, Rstandefer, Scerj, Simonwacker, SkyWalker, SlubGlub, Spurrymoses, Stephan Leeds, SteveLoughran, TEB728, Thumperward, Tobias Bergemann, Tomrbj, Whitehawk julie, WikiPuppies, 145 anonymous edits

Lazy systematic unit testing *Source*: http://en.wikipedia.org/w/index.php?oldid=198751809 *Contributors*: AJHSimons, Andreas Kaufmann, RHaworth

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

xUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=440723788 *Contributors*: Ahoerstemeier, Andreas Kaufmann, BurntSky, Caesura, Chris Pickett, Damian Yerrick, Dvib, FlashSheridan, Furrykef, Green caterpillar, Jpalm 98, Kenyon, Khatru2, Kku, Kleb, Lasombra, LilHelpa, MBisanz, Mat i, MaxSem, MindSpringer, MrOllie, Nate Silva, Ori Peleg, Pagrashtak, Patrikj, Pengo, PhilippeAntras, Pinecar, Qef, RedWolf, Rhphillips, RudaMoura, Schwern, SebastianBergmann, Simonwacker, Slakr, Srittau, Tlroche, Uzume, Woohookitty, 69 anonymous edits

List of unit testing frameworks *Source*: http://en.wikipedia.org/w/index.php?oldid=445904040 *Contributors*: A-Evgeniy, AJHSimons, Akadruid, Alan0098, AliveFreeHappy, Andreas Kaufmann, AndreasBWagner, Andrey86, Andy Dingley, Anorthup, Antonylees, Arjayay, Arjenmarkus, Artem M. Pelenitsyn, Asashour, Asimjalis, Ates Goral, Autarch, Avantika789, Avi.kaye, BP, Basvodde, Bdicroce, Bdijkstra, Beetstra, Berny68, Bigwhite.cn, Billyoneal, Boemmels, Brandf, BrotherE, Burschik, C1vineoflife, Calréfa Wéná, Chompx, Chris Pickett, ChronoKinetic, Ckrahe, Codefly, CompSciStud4U, Cpunit root, Cruftcraft, Cybjit, Dalepres, Damieng, DaoKaioshin, DataWraith, David smallfield, Decatur-en, Diego Moya, Dlindqui, Donald Hosek, DrMiller, EagleFan, Ebar7207, Edward, Eeera, Ellissound, Eoinwoods, Erkan Yilmaz, Figureouturself, Fltoledo, FredericTorres, Furrykef, Fuzlyssa, GabiS, Gaurav, Generalov.sergey, Ggeldenhuys, Gpremer, GregoryCrosswhite, Grincho, Grshiplett, Gurdiga, Harrigan, Harryboyles, Hboutemy, Hlopetz, Holger.krekel, Ian-blumel, IceManBrazil, Icseaturtles, Ilya78, JLaTondre, James Hugard, JavaCS, Jdpipe, Jens Lüdemann, Jeremy.collins, Jevon, Jim Kring, Joelittlejohn, Johnuniq, Jokes Free4Me, Jrosdahl, Justatheory, Jvoegele, KAtremer, Kenguest, Kiranthorat, Kku, Kleb, Kristofer Karlsson, Kwiki, LDRA, Legalize, Loopology, MMSequeira, Maine3002, Mandarax, Marclevel3, Mark Renier, Markvp, Martin Moene, Mdkorhon, MebSter, MeekMark, Mengmeng, Metalim, MiguelMunoz, Mindmatrix, Mitmacher313, Mj1000, Mkarlesky, Morder, NagyLoutre, Nereocystis, Nick Number, Nimowy, Nlu, Norrby, Northgrove, ObjexxWiki, Oestape, Ospalh, Paddy3118, Pagrashtak, Papeschr, PensiveCoder, Pentapus, Pesto, Pgr94, Pinecar, Praseodymium, Prekageo, Ptrb, R'n'B, RalfHandl, RandalSchwartz, Ravidgemole, Rawoke, Rcunit, Rhphillips, Rjollos, Rmkeeble, Robkam, Ropata, Rsiman, Ryadav, SHIMODA Hiroshi, Saalam123, Sarvilive, Schwern, Senfo, Sgould, Shabbychef, Shadriner, Siffert, Simeonfs, Simoneau, Simonscarfe, SirGeek CSP, Skiwi, Slhynju, Squares, Stassats, Stenyak, SteveLoughran, SummerWithMorons, Sutirthadatta, Swtechwr, Sydevelopments, Tabletop, Tadpole9, Tarvaina, Tassedethe, TempestSA, Ten0s, ThomasAagaardJensen, Thv, TobyFernsler, Tognopop, Torsknod, Traviscj, Uniwalk, Updatehelper, User77764, Uzume, Vassilvk, Vcmpk, Vibhuti.amit, Virtualblackfox, Wdevauld, Weitzman, Wernight, Wickorama, Winterst, Wodka, X!, Yince, Yipdw, Yukoba, Yurik, Zanhsieh, Zootm, Александр Чуранов, 538 anonymous edits

SUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=417107221 *Contributors*: Andreas Kaufmann, Chris Pickett, D6, Diegof79, Djmckee1, Frank Shearar, HenryHayes, Hooperbloob, Jerryobject, Mcsee, Nigosh, Olekva, TheParanoidOne, 7 anonymous edits

JUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=444269557 *Contributors*: 194.237.150.xxx, Abelson, AliveFreeHappy, Andmatt, Andreas Kaufmann, Andy Dingley, Anomen, Antonielly, Artaxiad, Ashwinikvp, Ausir, BeauMartinez, Biyer, Bluerasberry, Byj2000, Cat Parade, Cmdrjameson, Conversion script, DONOVAN, DaoKaioshin, Darc, Darth Panda, Doug Bell, Dsaff, Duplicity, East718, Epbr123, Esminis, Eye of slink, Faisal.akeel, Frap, Frecklefoot, Free Software Knight, Ftiercel, Furrykef, Ghostkadost, Gracenotes, Green caterpillar, Grendelkhan, Harrisony, Hervegirod, Hooperbloob, Ilya, Iosif, J0506, JLaTondre, Jerryobject, Jpalm 98, KellyCoinGuy, Kenguest, Kenji Toyama, Kent Beck, Kleb, KuwarOnline, M4gnum0n, MER-C, Mahmutuludag, Manish85dave, Mark Renier, Matt Crypto, Mdediana, MrOllie, Nate Silva, Nigelj, Ntalamai, POajdbhf, PaulHurleyuk, Paulsharpe, Pbb, Pcap, Plasmafire, Poulpy, Pseudomonas, Quinntaylor, Randomalious, Raztus, RedWolf, Resurgent insurgent, Rich Farmbrough, RossPatterson, SF007, Salvan, Sandipk singh, Science4sail, Silvestre Zabala, SirGeek CSP, Softtest123, Softwaresavant, Stypex, TakuyaMurata, TerraFrost, Thumperward, Tikiwont, Tlroche, Torc2, Tumbarumba, Tweisbach, UkPaolo, VOGELLA, Vina, Vlad, WiseWoman, Yamla, 117 anonymous edits

CppUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=433062569 *Contributors*: Amenel, Andreas Kaufmann, Anthony Appleyard, Conrad Braam, DSParillo, DrMiller, Frap, Ike-bana, Lews Therin, Martin Rizzo, Mecanismo, Mgfz, Rjwilmsi, Sysuphos, TheParanoidOne, Tobias Bergemann, WereSpielChequers, Yanxiaowen, 14 anonymous edits

Test::More *Source*: http://en.wikipedia.org/w/index.php?oldid=411565404 *Contributors*: Dawynn, Mindmatrix, Pjf, Schwern, Tassedethe, Unforgiven24

NUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=436487083 *Contributors*: Abelson, Andreas Kaufmann, B0sh, Brianpeiris, CodeWonk, Cwbrandsma, Djmckee1, Gfinzer, Gypwage, Hadal, Hooperbloob, Hosamaly, Ike-bana, Jacosi, Jerryobject, Kellyselden, Largoplazo, Magioladitis, MaxSem, MicahElliott, NiccciN, Nigosh, NinjaCross, PaddyMcDonald, Pinecar, Pnewhook, RHaworth, Raztus, RedWolf, Reidhoch, Rodasmith, S.K., SamuelTheGhost, Sj, StefanPapp, Superm401, Sydevelopments, Thv, Tobias Bergemann, Toomuchsalt, Ulrich.b, Valodzka, Whpq, Zsinj, 53 anonymous edits

NUnitAsp *Source*: http://en.wikipedia.org/w/index.php?oldid=367902681 *Contributors*: Andreas Kaufmann, Djmckee1, Edward, GatoRaider, Hooperbloob, Root4(one), SummerWithMorons

csUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=443653598 *Contributors*: Andreas Kaufmann, Djmckee1, Free Software Knight, Jerryobject, MaxSem, Mengmeng, Stuartyeates, 2 anonymous edits

HtmlUnit *Source*: http://en.wikipedia.org/w/index.php?oldid=440171119 *Contributors*: Agentq314, Andreas Kaufmann, Asashour, DARTH SIDIOUS 2, Edward, Jj137, KAtremer, Lkesteloot, Mabdul, Mguillem, Nigelj, Tobias Bergemann, Zwilson14, 36 anonymous edits

Test automation *Source*: http://en.wikipedia.org/w/index.php?oldid=445305303 *Contributors*: 5nizza, 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, Dreftymac, Eaowens, EdwardMiller, Egivoni, ElfriedeDustin, Elipongo, Enoch the red, Excirial, Faris747, Ferpectionist, FlashSheridan, Flopsy Mopsy and Cottonmouth, Florian Huber, Fumitol, G0gogcsc300, Gaggarwal2000, Gherget, Gibs2001, Gmacgregor, Goutham, Grafen, Harobed, Hatch68, Helix84, Hesa, Heydaysoft, Hooperbloob, Hswiki, Hu12, Ixfd64, JASpencer, JamesBWatson, Jkoprax, Johnuniq, Jpg, Kumarsameer, Kuru, Ldimaggi, M4gnum0n, MC10, MER-C, Marasmusine, Mark Kilby, Marudubshinki, Matthewedwards, Mdanrel, MendipBlue, Michael Bernstein, Morrillonline, MrOllie, Nima.shahhosini, Nimowy, Notinasnaid, Octoferret, Ohnoitsjamie, OracleDBGuru, PeterBizz, Pfhjvb0, ProfessionalTST, Qatutor, Qlabs impetus, Qtpautomation, Qwyrxian, R'n'B, RHaworth, Radagast83, Radiant!, Radiostationary, Raghublr, Raymondlafourchette, Rich Farmbrough, RichardHoultz, Rickjpelleg, Rjwilmsi, Robertvan1, Robinson Weijman, Ryadav, Ryepie, SSmithNY, Sbono, ScottSteiner, Seaphoto, Shankar.sathiamurthi, Shijuraj, Shlomif, SoCalSuperEagle, Softwaretest1, Srideep TestPlant, Ssingaraju, SteveLoughran, Sundaramkumar, Swtechwr, Testautomator, Thv, Ttrevers, Tumaka, Tushar291081, Vadimka, Veledan, Versageek, Vogelt, Waikh, Walter Görlitz, Webbbbbbber, Winmacro, Wrp103, Yan Kuligin, ZachGT, Zorgon7, सरोज कुमार ढकाल, 305 anonymous edits

Test bench *Source*: http://en.wikipedia.org/w/index.php?oldid=437094164 *Contributors*: Abdull, Ali65, Amitgusain, Arch dude, Dolovis, E2eamon, FreplySpang, J. Sparrow, Joe Decker, Pinecar, Remotelysensed, Rich Farmbrough, Singamayya, Testbench, Tgruwell, 12 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 stubs *Source*: http://en.wikipedia.org/w/index.php?oldid=357487595 *Contributors*: Andreas Kaufmann, Chiefhuggybear, Christianvinter, Meridith K, Thisarticleisastub, Tomrbj, 1 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

Test automation framework *Source*: http://en.wikipedia.org/w/index.php?oldid=426570771 *Contributors*: Abdull, Aby74, Akr7577, Al95521, AliveFreeHappy, Andy Dingley, Anshooarora, Apparition11, Chrisbepost, Closedmouth, Drpaule, Excirial, Flopsy Mopsy and Cottonmouth, Gibs2001, Heydaysoft, Homfri, Iridescent, JamesBWatson, Jonathan Webley, LedgendGamer, Mitch Ames, Mountk2, Nalinnew, Oziransky, Paul dexxus, Peneh, PeterBizz, Pinecar, Qlabs impetus, RHaworth, Regancy42, Rsciaccio, Sachxn, Sbasan, SerejkaVS, SteveLoughran, Vishwas008, Walter Görlitz, West.andrew.g, 55 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, Rwwww, SAE1962, Sbono, Sean.co.za, Zaphodikus, 32 anonymous edits

Modularity-driven testing *Source*: http://en.wikipedia.org/w/index.php?oldid=318953592 *Contributors*: Avalon, Minnaert, Phanisrikar, Pinecar, 2 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, Jtowler, Ken g6, Lowmagnet, Maguschen, MrOllie, Phanisrikar, Pinecar, Rjwilmsi, Rwwww, SAE1962, Scraimer, Sean.co.za, Sparrowman980, Swtesterinca, Ukkuru, Ultimus, Yun-Yuuzhan (lost password), Zoobeerhall, 57 anonymous edits

Hybrid testing *Source*: http://en.wikipedia.org/w/index.php?oldid=366545628 *Contributors*: Bunnyhop11, Horologium, MrOllie, Vishwas008, 5 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

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

Test-driven development *Source*: http://en.wikipedia.org/w/index.php?oldid=445516746 *Contributors*: 1sraghavan, Achorny, AliveFreeHappy, Alksentrs, Anorthup, AnthonySteele, Antonielly, Asgeirn, Astaines, Attilios, Autarch, AutumnSnow, Bcwhite, Beland, CFMWiki1, Calréfa Wéná, Canterbury Tail, Chris Pickett, Closedmouth, Craig Stuntz, CraigTreptow, DHGarrette, Dally Horton, David-Sarah Hopwood, Deuxpi, Dhdblues, Dougluce, Download, Downsize43, Dtmilano, Długosz, Ed Poor, Edaelon, Ehheh, Emurphy42, Enochlau, Eurleif, Excirial, Faught, Fbeppler, Fre0n, Furrykef, Gakrivas, 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, Jrvz, 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, Middayexpress, Mkarlesky, Mkksingha, Mnorbury, Mortense, Mosquitopsu, Mossd, Mr2001, MrOllie, Nigelj, Nohat, Notnoisy, Nuggetboy, Ojcit, Oligomous, On5deu, Parklandspanaway, Patrickdepinguin, Pengo, PhilipR, Phlip2005, Pinecar, PradeepArya1109, R. S. Shaw, Radak, Raghunathan.george, RickBeton, RoyOsherove, Rulesdoc, SAE1962, Sam Hocevar, Samwashburn3, San chako, Sanchom, SethTisue, Shadowjams, SharShar, Shenme, Shyam 48, SimonP, St.General, Stemcd, SteveLoughran, Sullivan.t, Supreme Deliciousness, Sverdrup, Svick, Swasden, Szwejkc, TakuyaMurata, Tedickey, Themacboy, Thumperward, Tobias Bergemann, Topping, Trum123, Underpants, V6Zi34, Virgiltrasca, WLU, Walter Görlitz, Waratah, Wikid77, Xagronaut, Олександр Кравчук, 408 anonymous edits

Agile testing *Source*: http://en.wikipedia.org/w/index.php?oldid=442661086 *Contributors*: Agiletesting, Alanbly, Athought, Chowbok, Eewild, Ehendrickson, Ericholmstrom, GoingBatty, Gurch, Hemnath18, Henri662, Janetgregoryca, Johnuniq, 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

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

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

Manual testing *Source*: http://en.wikipedia.org/w/index.php?oldid=437530057 *Contributors*: ArielGold, Ashish.aggrawal17, DARTH SIDIOUS 2, Donperk, Eewild, Hairhorn, Iridescent, Kgarima, L Kensington, Meetusingh, Morrillonline, OlEnglish, Pinecar, Pinethicket, Predatoraction, Rwxrwxrwx, Saurabha5, Somdeb Chakraborty, SwisterTwister, Tumaka, 53 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, Anorthup, 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, G0gogcsc300, 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, Neilc, 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

Ad hoc testing *Source*: http://en.wikipedia.org/w/index.php?oldid=443007678 *Contributors*: DRogers, Epim, Erkan Yilmaz, Faught, IQDave, Josh Parris, Ottawa4ever, Pankajkittu, Pinecar, Pmod, Robinson weijman, Solde, Walter Görlitz, 12 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, Histrion, Itai, JForget, Kaimiddleton, Karada, Kingpin13, LeaW, Lechatjaune, Lee Daniel Crocker, Martinwguy, Matma Rex, Melchoir, Mikewalk, Mild Bill Hiccup, NeilFraser, Nunh-huh, Oboler, PierreAbbat, Pinecar, Pinethicket, Polluks, R'n'B, Ricardol, Rrburke, Saberwyn, Sietse Snel, SimonTrew, Strait, Stratadrake, UlrichAAB, Verloren, Viriditas, Walter Görlitz, Webinfoonline, Wikid77, 98 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, Gggh, Gilliam, GreatWhiteNortherner, Hooperbloob, J.delanoy, Jewbacca, Jiang, Jtowler, Kmerenkov, Krashlandon, 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

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, Suffusion of Yellow, SusanLarson, Thv, Tmopkisn, Vishwas008, Vmahi9, Walter Görlitz, Wchkwok, Zhenqinli, 128 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, Rwwww, Walter Görlitz, 31 anonymous edits

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, Dhollm, Divyadeepsharma, Djmckee1, Dlevy-telerik, Eloquence, Emilybache, Enochlau, GTBacchus, GraemeL, Granburguesa, Gwernol, Halovivek, Hooperbloob, Hu12, Hutcher, Hyad, Infrablue, Jamestochter, Jemtreadwell, Jgladding, JimJavascript, Jmarranz, Jpp, Kaitanen, Ksnow, Liftoph, MartinDK, MeijdenB, Meise, Michael Hardy, Midnightcomm, Mifter, Mike Rosoft, Mjemmeson, Mortense, Mpilaeten, 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

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

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

Tester driven development *Source*: http://en.wikipedia.org/w/index.php?oldid=340119755 *Contributors*: Arneeiri, BirgitteSB, Chris Pickett, Fram, Gdavidp, Int19h, Josh Parris, Mdd, Pinecar, Sardanaphalus, Smjg, 11 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

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-verve, Pinecar, Pmberry, Robertvan1, Shizhao, Utuado, Walter Görlitz, 37 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 plan *Source*: http://en.wikipedia.org/w/index.php?oldid=440831109 *Contributors*: -Ril-, Aaronbrick, Aecis, Alynna 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, Jlao04, 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, Roshanoinam, Rror, SWAdair, Schmiteye, Scope creep, Shadowjams, SimonP, Stephenb, Tgeairn, The Thing That Should Not Be, Thunderwing, Thv, Uncle Dick, Wacko, Waggers, Walter Görlitz, Yparedes, 321 anonymous edits

Traceability matrix *Source*: http://en.wikipedia.org/w/index.php?oldid=446439560 *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, Timneu22, Voyagerfan5761, Walter Görlitz, WikiTome, 80 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, Hooperbloob, Iggy402, 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, Sciurinæ, Sean D Martin, Shervinafshar, Srikaaa123, Suruena, System21, Thejesh.cg, Thorncrag, Thv, Tomaxer, Travelbird, Velella, Vikasbucha, Walter Görlitz, Wernight, Yennth, 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 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 script *Source*: http://en.wikipedia.org/w/index.php?oldid=396053008 *Contributors*: Alai, Eewild, Falterion, Freek Verkerk, Hooperbloob, JLaTondre, JnRouvignac, Jruuska, Jwoodger, Michig, PaulMEdwards, Pfhjvb0, Pinecar, RJFJR, Rchandra, Redrocket, Sean.co.za, Sujaikareik, Teiresias, Thv, Ub, Walter Görlitz, 28 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

Static testing *Source*: http://en.wikipedia.org/w/index.php?oldid=433637055 *Contributors*: Aldur42, Amberved, Andreas Kaufmann, Avenue X at Cicero, Carlo.milanesi, Chris Pickett, Epim, Erkan Yilmaz, Iflapp, Iq9, Kauczuk, Nla128, Pinecar, Railwayfan2005, Rnsanchez, Robert Skyhawk, Ruud Koot, Sripradha, Walter Görlitz, Yaris678, 26 anonymous edits

Software review *Source*: http://en.wikipedia.org/w/index.php?oldid=433776641 *Contributors*: A Nobody, AliveFreeHappy, Andreas Kaufmann, Audriusa, Bovineone, Colonel Warden, Dima1, Donmillion, Gail, Irfibwp, Karada, Madjidi, Matchups, Rolf acker, Tassedethe, William M. Connolley, Woohookitty, XLerate, 33 anonymous edits

Software peer review *Source*: http://en.wikipedia.org/w/index.php?oldid=385388081 *Contributors*: AliveFreeHappy, Andreas Kaufmann, Anonymous101, Bovineone, Donmillion, Ed Brey, Ed Poor, Gronky, Karada, Kezz90, Kjenks, Lauri.pirttiaho, MarkKozel, Michael Hardy, Sdornan, Zakahori, 10 anonymous edits

Software audit review *Source*: http://en.wikipedia.org/w/index.php?oldid=416400975 *Contributors*: Andreas Kaufmann, Donmillion, JaGa, Katharineamy, Kralizec!, Romain Jouvet, Tregoweth, Woohookitty, Zro, 6 anonymous edits

Software technical review *Source*: http://en.wikipedia.org/w/index.php?oldid=405873452 *Contributors*: Andreas Kaufmann, Donmillion, Edward, Gnewf, 4 anonymous edits

Management review *Source*: http://en.wikipedia.org/w/index.php?oldid=398341890 *Contributors*: Andreas Kaufmann, Anticipation of a New Lover's Arrival, The, Ardric47, BagpipingScotsman, Bovineone, Deckiller, Donmillion, Galena11, JustinHagstrom, Karada, Outlook, Rintrah, 4 anonymous edits

Software inspection *Source*: http://en.wikipedia.org/w/index.php?oldid=442930498 *Contributors*: A.R., Andreas Kaufmann, AndrewStellman, Arminius, AutumnSnow, BigMikeW, Bigbluefish, Bovlb, Ebde, Ft1, Fuzheado, Ivan Pozdeev, Kku, Michaelbusch, Mtilli, Nmcou, Occono, PeterNuernberg, Rmallins, Secdio, Stephenb, SteveLoughran, Vivio Testarossa, Wik, 56 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, Epeefleche, Gaff, Gaius Cornelius, Gimmetrow, Hockeyc, Icarusgeek, Iwearavolcomhat, JIP, Kezz90, MacGyverMagic, Mjevans, Mkjadhav, Nick Number, Okok, Pedro.haruo, Slightsmile, Tagishsimon, Talkaboutquality, Tassedethe, The Font, The Letter J, Zundark, 43 anonymous edits

Software walkthrough *Source*: http://en.wikipedia.org/w/index.php?oldid=402654664 *Contributors*: Andreas Kaufmann, DanielPharos, Donmillion, Gnewf, Jherm, Jocoder, Karafias, Ken g6, OriolBonjochGassol, Reyk, 12 anonymous edits

Code review *Source*: http://en.wikipedia.org/w/index.php?oldid=436620210 *Contributors*: Adange, Aivosto, AliveFreeHappy, Alla tedesca, Andreas Kaufmann, Bevo, Brucevdk, Cander0000, CanisRufus, ChipX86, Craigwb, DanielVale, Derek farn, Digsav, Dwheeler, Ed Poor, Enigmasoldier, Flamurai, Fnegroni, Furrykef, Gbolton, Hooperbloob, Intgr, J.delanoy, Jabraham mw, Jamelan, Jesselong, Khalid hassani, Kirian, Kispa, Madjidi, Martinig, Matchups, MattOConnor, MattiasAndersson, MrOllie, Mratzloff, Msabramo, Mutilin, NateEag, Nevware, Oneiros, Pcb21, Project2501a, Pvlasov, Rajeshd, Ronz, Rrobason, Ryguasu, Salix alba, Scottb1978, Smartbear, Srice13, StefanVanDerWalt, Steleki, Stephenb, Stevietheman, Sverdrup, Swtechwr, Talkaboutquality, Themfromspace, ThurnerRupert, Tlaresch, Tom-, Ynhockey, ZamorakO o, 106 anonymous edits

Automated code review *Source*: http://en.wikipedia.org/w/index.php?oldid=440742305 *Contributors*: Aivosto, AliveFreeHappy, Amoore, Andreas Kaufmann, Closedmouth, Download, Elliot Shank, Fehnker, Gaudol, HelloAnnyong, IO Device, JLaTondre, Jabraham mw, John Vandenberg, Leolaursen, Lmerwin, Mellery, NathanoNL, OtherMichael, Pgr94, Ptrb, Pvlasov, RedWolf, Rwwww, Swtechwr, ThaddeusB, Tracerbee, Wknight94, 27 anonymous edits

Code reviewing software *Source*: http://en.wikipedia.org/w/index.php?oldid=394379382 *Contributors*: Aivosto, AliveFreeHappy, Amoore, Andreas Kaufmann, Closedmouth, Download, Elliot Shank, Fehnker, Gaudol, HelloAnnyong, IO Device, JLaTondre, Jabraham mw, John Vandenberg, Leolaursen, Lmerwin, Mellery, NathanoNL, OtherMichael, Pgr94, Ptrb, Pvlasov, RedWolf, Rwwww, Swtechwr, ThaddeusB, Tracerbee, Wknight94, 27 anonymous edits

Static code analysis *Source*: http://en.wikipedia.org/w/index.php?oldid=410764388 *Contributors*: 212.153.190.xxx, A5b, Ahoerstemeier, Alex, AliveFreeHappy, Andareed, Andreas Kaufmann, Antonielly, Anujgoyal, Berrinam, CaliforniaAliBaba, Conversion script, Creando, Crowfeather, Cryptic, DatabACE, David.Monniaux, Dbelhumeur02, Dekisugi, Derek farn, Diego Moya, Ebde, Ed Brey, Erkan Yilmaz, Fderepas, Ferengi, FlashSheridan, Gadfium, Goffrie, GraemeL, Graham87, Ground Zero, Ixfd64, JForget, Jabraham mw, JacobTrue, Jan1nad, Jisunjang, JoelSherrill, JohnGDrever, Jpbowen, Jschlosser, Kazvorpal, Ks0stm, Kskyj, Leibniz, Marudubshinki, Mike Van Emmerik, Mutilin, Peter M Gerdes, Pinecar, Ptrb, Qwertyus, Renox, Rpm, Ruud Koot, Rwwww, Sashakir, Schwallex, StaticCast, Sttaft, Suruena, Swtechwr, TUF-KAT, Ted Longstaffe, Thv, Tinus74, Tjarrett, Tregoweth, Villeez, Vina, Vkuncak, Vp, Wlievens, Yonkie, 111 anonymous edits

List of tools for static code analysis *Source*: http://en.wikipedia.org/w/index.php?oldid=446462849 *Contributors*: 70x7plus1, A.zitzewitz, Achituv, Adarw, Aetheling, Aivosto, Albert688, Alex, Alexcenthousiast, Alexius08, Alextelea, AliveFreeHappy, Amette, Amire80, Amoore, Andreas Kaufmann, AndrewHowse, Angusmclellan, Apcman, Ariefwn, Armadillo-eleven, Athaenara, Atif.hussain, Avraham, Azrael Nightwalker, BB-Froggy, Bakotat, Bantoo12, Bchess, Bdoserror, Bellingard, Benneman, Benrick, Bensonwu, Bgi, Bjcosta, Bknittel, Bkuhn, Bnmike, Borishollas, Breakpoint, Camwik75, Capi x, Catamorphism, Cate, Cgisquet, Checkshirt, Chick Bowen, Collinpark, Cpparchitect, Cryptic, CxQL, Dash, DatabACE, David Gerard, David.Monniaux, Dbelhumeur02, Dekisugi, Demarant, Derek farn, Devpitcher, Diego Moya, Dinis.Cruz, Diomidis Spinellis, Disavian, Dmkean, Dmooney, Dmulter, Dnozay, DomQ, Donaldsbell@yahoo.com, Douglaska, Dpnew, Drdeee, Drpaule, Dtgriscom, Dvice null, Ed Brey, Ehajiyev, Elliot Shank, Epierrel, Esdev, Ettl.martin, Exatex, Excirial, Faganp, Falcon9x5, Felmon, FergusBolger, Fewaffles, Fishoak, Flamingcyanide, FlashSheridan, Fowlay, Frap, Freddy.mallet, FutureDomain, Fwaldman, G b hall, Gahs, Gaius Cornelius, Gaudol, Gbickford, Gesslein, Giggy, Gogege, Grauenwolf, Guillem.Bernat, Gwandoya, Haakon, Hello484, HelloAnnyong, Hooperbloob, Hyd danmar, Iceberg1414, Imeshev, InaToncheva, InaTonchevaToncheva, Irishguy, Issam lahlali, Istoyanov,

JLaTondre, Jabraham mw, Jamieayre, Jayabra17, Jayjg, Jcuk 2007, Jdabney, Jeodesic, Jersyko, Jisunjang, JnRouvignac, Joebeone, John of Reading, JohnGDrever, Jopa fan, Jpbowen, Jschlosser, Jsub, Kengell, Kenguest, Kent SofCheck, Kfhiejf6, Klausjansen, Kravietz, Krischik, Krlooney, Kskyj, LDRA, Lalb, Libouban, LilHelpa, Linehanjt, Llib xoc, Lmerwin, Malcolma, Mandrikov, Martarius, MartinMarcher, Matsgd, Mcculley, Mdjohns5, Mike Van Emmerik, Mikeblas, Minhyuk.kwon, Mj1000, Mmernex, Monathan, Moonwolf14, Mrlongleg, Mrwojo, N5iln, Nandorjozsef, Nandotamu, Nbougalis, Neerajsangal, NewSkool, Newtang, Nick Number, Nickj, Nico.anquetil, Nixeagle, Northgrove, Notopia, O2user, Oorang, Optimyth, Orangemike, PSeibert, PSmacchia, Parasoft-pl, Parikshit Narkhede, PaulEremeeff, Pauljansen42, Pausch, Pavel Vozenilek, Perrella, Petdance, Pfunk1410, Phatom87, Piano non troppo, Pinecar, Pitkelevo, Pkortve, Pkuczynski, Pmjtoca, Pmollins, Pokeypokes, Prasanna vps, PraveenNet, Pth81, Ptrb, Pvlasov, Qu3a, RHaworth, Rainco, Rajah9, Ralthor, Rdbuckley, Rhuuck, Rich Farmbrough, Richsz, RickScott, Rodolfo Borges, Romgerale, Rosen, Rpapo, Rpelisse, Rrtuckwell, Rssh, Runtime, Ruud Koot, Sachrist, Sadovnikov, Sander123, Sashakir, Schwallex, Scovetta, Serge Baranovsky, Sffubs, ShelfSkewed, Shiva.rock, Siva77, Skilner, Skrik69, Sourceanalysis, Sreich, Staniuk, StaticCast, Stephen.gorton, Sttaft, Stubb, Swtechwr, Taed, Tasc, Tddcodemaster, Tedickey, Test-tools, The.gaboo, Timekeeper77, Tjarrett, Tkvavle, Tlownie, Tomtheeditor, Tonygrout, Toutoune25, Traal, Tracerbee, Tradsud, Tregoweth, Uncopy, Vaucouleur, Velizar.vesselinov, Venkatreddyc, Verdatum, Verilog, Vfeditor, Vor4, Vp, Wakusei, Wdfarmer, Wegra, Weregerbil, Wesnerm, Wiki jmeno, Wikieditoroftoday, Wikimaf, Woohookitty, Wws, Xodlop, Xoloz, Yansky, Yoderj, Ysangkok, Zfalconz, 725 anonymous edits

GUI software testing *Source*: http://en.wikipedia.org/w/index.php?oldid=413588113 *Contributors*: 10metreh, Alexius08, AliveFreeHappy, Andreas Kaufmann, Chaser, Cmbay, Craigwb, Dreftymac, Dru of Id, Equatin, Gururajs, Hardburn, Hesa, Hu12, Imroy, Jeff G., JnRouvignac, Josephtate, Jruuska, Jwoodger, Ken g6, Liberatus, MER-C, Mcristinel, Mdjohns5, Mild Bill Hiccup, Paul6feet1, Pinecar, Pnm, Rdancer, Rich Farmbrough, Rjwilmsi, Rockfang, Ronz, SAE1962, SiriusDG, Staceyeschneider, SteveLoughran, Steven Zhang, Unforgettableid, Wahab80, Wakusei, 54 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, Antariksawan, Arthena, Azrael81, Bihco, Bkillam, Brandon, Breakthru10technologies, Bretclement, ChrisJMoor, Christopher Agnew, Cjohansen, Ckatz, Conversion script, Crònica, DXBari, Dennis G. Jerz, Dickohead, Diego Moya, Dobrien, DrJohnBrooke, Dvandersluis, EagleFan, Farreaching, Fredcondo, Geoffsauer, Gmarinp, Gokusandwich, GraemeL, Gubbernet, Gumoz, Hede2000, Hooperbloob, Hstetter, JDBravo, JaGa, Jean-Frédéric, Jetuusp, Jhouckwh, Jmike80, Karl smith, Kolyma, Kuru, Lakeworks, Leonard^Bloom, LizardWizard, Mandalaz, Manika, MaxHund, Mchalil, Miamichic, Michael Hardy, MichaelMcGuffin, MikeBlockQuickBooksCPA, Millahnna, Mindmatrix, Omegatron, Pavel Vozenilek, Philipumd, Pigsonthewing, Pindakaas, Pinecar, QualMod, Ravialluru, Researcher1999, Rich Farmbrough, Rlsheehan, Ronz, Rossami, Schmettow, Shadowjams, Siddhi, Spalding, Tamarkot, Technopat, Tobias Bergemann, Toghome, Tomhab, Toomuchwork, Vmahi9, Wikinstone, Wikitonic, Willem-Paul, Wwheeler, Yettie0711, ZeroOne, 129 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

Usability inspection *Source*: http://en.wikipedia.org/w/index.php?oldid=382541633 *Contributors*: Andreas Kaufmann, Diego Moya, Lakeworks, 2 anonymous edits

Cognitive walkthrough *Source*: http://en.wikipedia.org/w/index.php?oldid=440231930 *Contributors*: American Eagle, Andreas Kaufmann, Avillia, Beta m, DXBari, David Eppstein, Diego Moya, Elusive Pete, Firsfron, FrancoisJordaan, Gene Nygaard, Karada, Kevin B12, Lakeworks, Macdorman, Masran Silvaris, Moephan, Naerii, Quale, Rdrozd, Rich Farmbrough, SimonB1212, Spalding, Srbauer, SupperTina, Tassedethe, Vacarme, Xionbox, 33 anonymous edits

Heuristic evaluation *Source*: http://en.wikipedia.org/w/index.php?oldid=422883455 *Contributors*: 0403554d, Andreas Kaufmann, Angela, Art LaPella, Bigpinkthing, Catgut, Clayoquot, DXBari, DamienT, Delldot, Diego Moya, Edward, Fredcondo, Fyhuang, Hugh.glaser, JamesBWatson, Jonmmorgan, JulesH, Karada, KatieUM, Kjtobo, Lakeworks, Luiscarlosrubino, PhilippWeissenbacher, RichardF, Rjwilmsi, Ronz, SMasters, Subversive, Turadg, Verne Equinox, Wikip rhyre, Zeppomedio, 48 anonymous edits

Pluralistic walkthrough *Source*: http://en.wikipedia.org/w/index.php?oldid=440233676 *Contributors*: Andreas Kaufmann, Diego Moya, Lakeworks, Minnaert, RHaworth, Team Estonia, 4 anonymous edits

Comparison of usability evaluation methods *Source*: http://en.wikipedia.org/w/index.php?oldid=406062283 *Contributors*: Andreala, Diego Moya, Eastlaw, Lakeworks, RHaworth, Ronz, Simone.borsci, 3 anonymous edits

Image Sources, Licenses and 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: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:Three point flexural test.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Three_point_flexural_test.jpg *License*: Creative Commons Attribution-Sharealike 3.0,2.5,2.0,1.0 *Contributors*: Cjp24

File:US Navy 070409-N-3038W-002 Aviation Structural Mechanic 3rd Class Rene Tovar adjusts a connection point on a fixture hydraulic supply servo cylinder test station in the hydraulics shop aboard the Nimitz-class aircraft carrier U.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:US_Navy_070409-N-3038W-002_Aviation_Structural_Mechanic_3rd_Class_Rene_Tovar_adjusts_a_connection_point_on_a_fixture_hydraulic_supply_servo_cylinder_test_ *License*: Public Domain *Contributors*:

File:2009-0709-earthquake.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:2009-0709-earthquake.jpg *License*: Public Domain *Contributors*: Photo Credit: Colorado State University

File:US Navy 070804-N-1745W-122 A Sailor assigned to Aircraft Intermediate Maintenance Department (AIMD) tests an aircraft jet engine for defects while performing Jet Engine Test Instrumentation, (JETI) Certification-Engine Runs.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:US_Navy_070804-N-1745W-122_A_Sailor_assigned_to_Aircraft_Intermediate_Maintenance_Department_(AIMD)_tests_an_aircraft_jet_engine_for_defects_while_perform *License*: Public Domain *Contributors*:

File:TH11-50kN-pincer-grip.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:TH11-50kN-pincer-grip.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Ingeniero-aleman

File:THS527-50.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:THS527-50.jpg *License*: Creative Commons Attribution 3.0 *Contributors*: Ingeniero-aleman

File:TH-screw-grips.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:TH-screw-grips.jpg *License*: GNU Free Documentation License *Contributors*: Ingeniero-aleman

File:THS766-5.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:THS766-5.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

File:THS314-2.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:THS314-2.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

File:THS13k-02-200N.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:THS13k-02-200N.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

File:Temperaturkammer-spannzeug THS321-250-5.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Temperaturkammer-spannzeug_THS321-250-5.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

File:TH149 .jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:TH149_.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

File:THS137-4-fr.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:THS137-4-fr.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman http://www.grip.de

File:Biegevorrichtung TH165.jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Biegevorrichtung_TH165.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

File:Abzugsvorrichtung TH50+SW .jpg *Source*: http://en.wikipedia.org/w/index.php?title=File:Abzugsvorrichtung_TH50+SW_.jpg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ingeniero-aleman

Image:NUnit GUI.png *Source*: http://en.wikipedia.org/w/index.php?title=File:NUnit_GUI.png *License*: unknown *Contributors*: MaxSem

Image:CsUnit2.5Gui.png *Source*: http://en.wikipedia.org/w/index.php?title=File:CsUnit2.5Gui.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Manfred Lange

Image:htmlunit logo.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Htmlunit_logo.png *License*: Fair Use *Contributors*: Agentq314

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: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_du *License*: Public Domain *Contributors*:

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: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.

License

Creative Commons Attribution-Share Alike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/