

Application Test Design Patterns

Common Test Patterns

The usual test-design process is very repetitive, yet many testing problems have generic elements that you can reuse easily in creation of later test designs. You can capture these patterns, and use them to simplify the creation of later test designs.

One problem that still limits effectiveness and efficiency in the testing of applications is that a usual development-and-test cycle involves a lot of creation of test-design details from scratch, and in a somewhat ad-hoc manner, whenever a new set of application features is defined. The primary tool often is an empty test-design template, which includes categories of test issues to address enumeration of test specifications or test-case details that are based on feature functional specifications.

If viewed across a series of project cycles, test-design activities are one area in which there is a lot of repetition: Test engineers start with a blank page in a test-design specification and fill it in detail by detail, when many of these issues are generic and common to nearly every application-testing scenario.

A goal of the test-design process should be to reduce ad-hoc reinvention of test designs by capturing and reusing common test patterns.

The recognition of test patterns is loosely similar to design patterns that are used in software development. (There are also some classic object-oriented design patterns that might be suitable for test-automation design, but we do not address those patterns directly.) Creating test designs by reusing patterns can cover repetitive, common core verification of many aspects of an application. This frees the test engineer to focus more time and thought on truly unique aspects that are particular to an application without having to waste effort on definition of the more generic portions of the test design.

Just as developers who are writing software strive to minimize the code that they reinvent or create from scratch by using existing functionality for common, generic, or repeated operations, so too should test engineers strive to minimize the amount of time they spend on defining test cases and verification steps for functionality that is common to every application with which they are presented. Test engineers should be able to reuse the full and rich set of tests that previously have been identified for similar situations.

Expert-Tester Pattern

Experienced, expert testers often approach these issues by applying their knowledge and past experience as heuristics to recognize what test situations align with techniques, test issues, and test approaches. When they see a given test situation, they immediately identify a method of testing that is appropriate and applicable. Novices, on the other hand, are likely to be unaware of all of the test issues or pitfalls that are associated with the situation or the test conditions that should be applied. In addition, expert knowledge is not captured explicitly into a highly transferrable format for the novice to be able to take advantage of it, and it often can be lost to the team when the expert has moved on.

Test-Category Checklists

An underused tool that provides the first step in capturing product-testing experience is the checklist. Checklists make the test process and accumulated ideas explicit for reuse at a later time by the same or another test engineer. Often, they are lists of heuristics and mnemonic devices that should be applied to the testing and verification process, to avoid forgetting any obvious tests. Effective checklists in this context are abstracted so that they are applicable to multiple projects or applications. The appropriate level of abstraction is to: Check each entry field that has maximum-length string data.

- Check each required field that has null data.
- Challenge type-checking validation of each field that has data of a different type from what is allowed.
- Test for all data values that are simultaneously maximized or most stressful.
- Verify that the nominal value of data is accepted, stored, retrieved, and displayed.

Unfortunately, generic and feature-specific test objectives often are jumbled together, without thought being given to reuse; the generic aspects of the design are buried in the specifics, and the value for reuse is reduced.

Failure Modes

Successful tests are ones that find problems; the problems that are found often are due to timeless mistakes and common failure modes. Some problem areas have classic errors in implementation; some conceptual designs have classic mistakes that are associated with them; some technologies have inherent flaws or limitations; some failures are critical or important, and must be validated; and some error-handling consistently is neglected.

The test-design processes of expert testers often rely, in part, on experience about things that have gone wrong in the past.

However, this past experience is not included in any functional specification for a product, and therefore is not called out explicitly to verify. The experience of an expert tester can be captured and turned into valuable test-design heuristics. Testing is organized often around failure modes, so that enumeration of failure modes is a valuable portion of test design. Lack of anticipation of new, novel, or infrequent failure modes is the prime source of test-escapement problems, where bugs are not detected during testing phases, but affect customers who use the application. Often, these bugs are the result of failure modes that could have been anticipated—based upon past product-testing experience—but the experience was not captured, shared, or passed on to a novice tester. An easy way to avoid this problem is to write generic specifications for common failure modes, convert them into test conditions to verify or validate, and concatenate this to any test-design template.

Enumeration of Things to "Cover"

A useful approach to the multidimensional, orthogonal nature of test planning is to identify categories of tests that have strong conceptual relationships, and then use those subsets to ensure that testing completeness is achieved in each category. One concept that is helpful is to break the problem into distinct manageable categories, based on the idea of "coverage." In this case, the concept of coverage can be applied in a more generic, broadly applicable form than the specific techniques that are used to identify untested areas of the code (such as the metrics that are collected with code-coverage toolsets).

The key part of this analysis process is to create as many separate categories of things that can be "covered" as possible. Iterate through each category, to build lists of test items that should be covered. Common categories to use as a starting point for the process are user-scenario coverage, specification-requirement coverage, feature coverage, UI control coverage, field coverage, form coverage, output/report coverage, error coverage, code coverage, condition coverage, and event coverage.

The concept can be extended to many other areas. Identification of a number of test categories aids the analysis and design process that often is applied to requirements review for test identification. Having a formalized set of categories allows the test engineer to decompose requirements into testable form, by dividing the problem into testable fragments or test matrices across a number of dimensions.

Generic Test-Design Templates

Test-design specification templates should be expanded beyond the typical empty-document sections to fill in, to include general validation rules, test-case checklists, common failure modes, and libraries of generic tests. The templates should include many of the usual tests for application features, so that they do not have to be added or reinvented. Part of this generic design should implement "boilerplate" tests that will be the first ones that every testing effort should include, even before considering more sophisticated and instance-specific tests. In this way, the generic test-design specifications allow any tester—for example, a new contingent staff member—to perform a range of generic tests, even before any customized, application-specific test planning.

Pesticide Paradox

Test automation—especially, when it is applied (as usual default) as regression test automation—suffers from static test data. Without variation in the data or execution path, bugs are located with decreasing frequency. The paradox is that the reduction in bugs that are found per test pass is not representative of the total number of bugs that remain in the system. Bugs are only less common on the well-exercised code paths that are represented by the static data. Bugs that remain in untested areas of the application are resistant to the testing methods (the automation pesticide paradox).

The lack of variation sometimes can improve the stability of the test code somewhat, but at the detriment of good test coverage. Effectiveness of testing—as measured in the detection of new bugs—is improved significantly by the addition of variation, such as a test pass using different values of data within the same equivalence class. Automation should be designed so that a test-case variable can provide a common equivalence class or data type as a parameter, and the automation framework will vary the data randomly within that class and apply it for each test run. (The automation framework should record the seed values or actual data that was used, to allow reruns and retesting with the same data for debugging purposes.)

In some cases, working with dynamically selected input data can be problematic because determining the expected outcome may be difficult (the Oracle problem). However, in many cases, there are tests that can be selected and verified automatically, even without requiring sophisticated approaches to determine the correct test output for a particular input.

Test-automation strategies too often are based solely on the automation of regression tests, putting all of the eggs in that basket alone. This is analogous to buying insurance to protect against failures that are due to regression: It is expensive, and it mitigates failures in the areas that the insurance (automation) covers, but it does not reduce or prevent the problem in uninsured (unautomated) portions of the application. Wise insurance-purchasing strategies usually are based on the selection of the optimal amount, and not on the spending of all available resources on a gold-plated, zero-deductible insurance policy. In testing terms, some teams spend too much time on automation—at the expense of lost mindshare that could be dedicated to other types of testing, and putting their projects at increased risk of detecting only a small portion of the real problems.

The cost/benefit ratio of test automation can be tilted in a positive direction by concentrating on adding stability to development processes, identifying sources of regression failures, and fixing them systematically. When pursued with vigor, it soon becomes inappropriate to over-invest in a regression test-automation strategy, and much more important to invest in other more leveraged

forms of test automation. Otherwise, the dominant problem changes quickly from creating tests to find bugs to trying to manage and maintain myriad redundant, overlapping, obsolete, or incorrect regression test cases.

A regression-focused approach can result in automated tests that are too static. The cost/benefit ratio is marginal for these tests when all of the maintenance and management costs are taken into account; they do not find many bugs. If test systems are designed to challenge dynamically the full range of data-dependent failures, alternate invocation modes, and number of iterations they can accomplish much more than regression testing.

By increasing the abstraction of test automation to "find all strings on the form and test them with all generic tests (apply random data variations until an error occurs or 100,000 conditions are tried)" and running the tests continually, testing is elevated from a marginally valuable static regression suite to a generic tool that has a much higher probability of adding value to the project. (It has the added benefit of providing technical opportunities for extending Software Test Engineers, instead of just chaining their desk to maintain legacy automation.)

The other benefit of higher-level abstraction and making tests more generic is efficiency; adding test cases should be simple and quick, and not require hours of custom test-case coding of minutiae. Having test automation designed at the level of "test this control()"—instead of 50 lines of "click here, type this, wait for this window"—empowers a broad range of testers to automate tests and focus efforts on planning meaningful tests, instead of having to slog through the implementation details of simplistic scenarios that are coded against poorly abstracted test frameworks.

Conclusion

Testers should build test designs around reusable test patterns that are common to a large number of application-testing problems. The inclusion of these patterns as a standard part of the test-design template reduces the time that is spent on test design, by eliminating the need to start with a blank test-design specification. Reuse of these patterns has the added benefit of codifying expert-tester knowledge, so as to increase the likelihood of catching common failures.

source: Mark Folkerts, Tim Lamey, and John Evans - <http://msdn.microsoft.com/en-us/testing/cc514239>.

Application Test Design Patterns

Reuse Test Patterns

There are many useful approaches that are being applied to well-designed tests and can contribute to the reuse of test design:

Equivalence-Class Partitioning

Boundary-Value Conditions

Strong Data Typing

Design Patterns

Model-Based Testing

Test Oracles

Test-Data Generators

One of the benefits of reusable test design is that you can take advantage of the convergence of these concepts and apply the results, so that they improve the cost/benefit ratio of test efforts on future projects. All of these techniques have elements that contribute to raising test-design abstraction—improving reuse, reducing reinvention, elevating the sophistication of tests, and extending the reach of Software Test Engineers.

Test-Design Reuse Summary

- Reduce ad-hoc reinvention of test designs by capturing and reusing common test patterns.
- Recognize and abstract test issues, and capture them in a form that can be reapplied as a higher level chunk of activity, instead of always dropping down to the detailed instance level.
- Treat past test designs essentially as individual instantiations of a generic test-design "class."
- Institutionalize, "package," or "productize" the results and experience of past test-design work into a reusable deliverable that can be applied to similar test efforts.
- Create checklists of generalized "things to test" that will be a resource for reuse on future versions or applications. These generalized tests augment the separate list of feature details (most often, business rules) and associated tests that truly are unique to a specific project.
- Enumerate common and generic failure modes as items to verify, and make them part of test-design templates.

- Enumerate and document explicit test "coverage" categories, so that they can be incorporated into analysis and test design.
- Ensure that each category contains as complete a set of tests as possible to ensure coverage of the category. Divide the results into generic and unique subsets of tests. Reuse the generic portions of these lists for future efforts.
- Test-design specification templates should be expanded beyond empty-document sections to fill in, to include generic validation rules, test-case checklists, common failure modes, and libraries of generic tests.
- Avoid the pesticide paradox by using a test infrastructure that provides for flexible data-oriented testing at the equivalence-class or data-type level, instead of supporting only specific, handcrafted test cases. Varying the test data and paths of execution will increase the probability of detecting new bugs.

Basic Examples

The following are a few frequently occurring test-design patterns that are suitable for reuse.

CRUD Pattern

1. Identify a record or field upon which to operate (based on input name and parameter info).
2. Generate randomized item from equivalence classes.
3. Verify nonexistence.
4. Add item.
5. Read and verify existence of identical unchanged data.
6. Modify and verify matching modified data.
7. Delete and verify removal of item.

Data-Type Validation Pattern

1. Identify item with type characteristics (for example, a data field) at an abstract level; this should not be limited to simple data types, but should include common business data types (for example, telephone number, address, ZIP code, customer Social Security number, calendar date, and so on).
2. Enumerate the generic business rules that are associated with the type.
3. Define equivalence partitions and boundaries for the values for each business rule.
4. Select test-case values from each equivalence class.

Input Boundary Partition Pattern

1. Enumerate and select an input item.
2. Select a "valid" equivalence partition.
3. Apply a lookup or random generation of a value within that partition, and use it for test.

Stress-Axis Pattern

1. Identify a system response to verify.
2. Identify a stress axis (for example, number of items, size of items, frequency of request, concurrency of requests, complexity of data structures, and dimensionality of input).
3. Identify a starting level of stress.
4. Verify and measure system response.
5. Increase stress, and repeat cycle until system response fails.
6. Switch to another stress axis.
7. Increase the level of that axis until failure.
8. Additionally, add concurrent stress axes.
9. Increase the number of concurrent axes until failure.

Heuristic of Error-Mode Detection Pattern

1. Enumerate past human-error modes.
2. Select a mode that has observed recurrence.
3. Identify a scope in which the failure mode might apply, and routinely test for that failure until you are convinced that it manifested.

File Test Pattern

1. Identify and define files and file semantics to be evaluated.
2. Enumerate failure modes for files.
3. Identify system response for each failure mode to verify (create an Oracle, list).
4. Select each failure mode, and apply it in turn to the designated file.
5. Verify response.

Generic Window UI Operations

1. Open/Close/Maximize/Minimize.
2. Active/Inactive.
3. Validate modality.
4. Validate proper display and behavior when on top or behind.
5. Expand/Resize.
6. Move.

Conclusion

Testers should build test designs around reusable test patterns that are common to a large number of application-testing problems. The inclusion of these patterns as a standard part of the test-design template reduces the time that is spent on test design, by eliminating the need to start with a blank test-design specification. Reuse of these patterns has the added benefit of codifying expert-tester knowledge, so as to increase the likelihood of catching common failures.

source: Mark Folkerts, Tim Lamey, and John Evans - <http://msdn.microsoft.com/en-us/testing/cc514239>.

Application Test Design Patterns

Application Scope Testing Patterns

Extended Use Case Test Pattern

Intent: Develop an application system test suite by modeling essential capabilities as extended use cases.

Context: Extended Use Case test applies when most, if not all, of the essential requirements for the System Under Test can be modeled. Use cases are not always the model of choice for essential system scope capabilities. For example, consider a complex simulation that renders the results with animated, high resolution graphics. The setup and inquiry for a simulation run could be modeled with use cases. However, these use cases could only hint at the essential requirements. It is possible to develop test cases by imagining specific instances for a use case and corresponding expected results.... More in

source:

Bob Binder - Testing Object-oriented Systems: Models, Patterns, and Tools

<http://www.rbsc.com/pages/TestDesignPatterns.html>

Model Driven Test Pattern

Name: MDT

Intent: Test Objectives

Context: Testing through an API, or with a test automation framework.

Fault Model: Complex sequencing faults

Strategy:

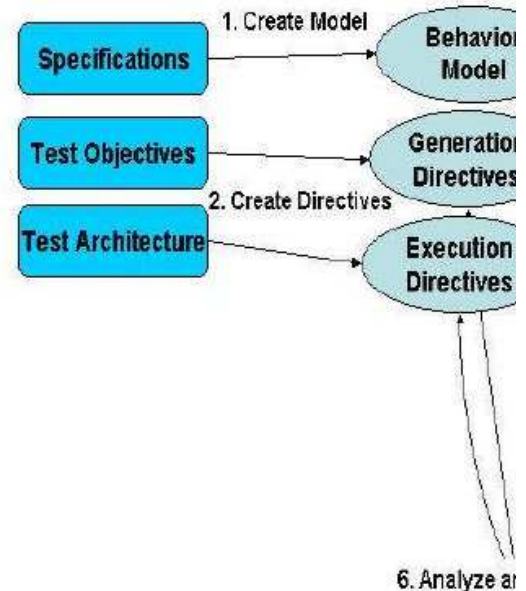
- Model: Step 1
- Procedure: Step 3
- Oracle: From Step 1
- Automate: From Test Architecture

Entry: Model and Architecture well defined

Exit: Analyze and Feedback

Consequences: Upfront investment, thorough testing,

Uses: Huge bibliography



source: Alan Hartman - http://www.imbus.de/qs-day/2007/vortraege/hartman_TestPatterns.pdf

Class Test Design Patterns

Class Scope Testing Patterns

Class scope testing presents test patterns for result-oriented testing at class scope. Class test design must consider method responsibilities, intraclass interactions, and superclass/subclass interactions.

Category-Partition Test Pattern

Intent: Design test suites for methods based on input-output relationships

Context: A systematic technique to identify test cases for a single method based on input-output pairs. Appropriate for cohesive functions that depend on multiple inputs to make decisions or computations.

More information: [TestPatternCategoryPartition.pdf](#)

source:

Bob Binder - Testing Object-oriented Systems: Models, Patterns, and Tools

<http://www.rbsc.com/pages/TestDesignPatterns.html>

Alan Hartman - http://www.imbus.de/qs-day/2007/vortraege/hartman_TestPatterns.pdf

Percolation Test Pattern

Intent: Implement automatic checking of superclass assertions to support design-by-contract and the Liskov substitution principle.

Context:The percolation pattern allows base class invariants, preconditions, and postconditions to be checked in derived class functions without the use of redundant code. It respects base class encapsulation by providing a fixed interface to base class assertions without direct access to base class instance variables. It allows loosely coupled, compile-time enabling/disabling of assertion checking.

More information: [Percolation.pdf](#)

source:

Bob Binder - Testing Object-oriented Systems: Models, Patterns, and Tools

<http://www.rbsc.com/pages/HarnessPatternList.htm>

Gang of Four Test Design Patterns

Gang of Four Test Patterns

The Observer Test Pattern

A portion of the design of an application has been created using the Observer pattern. Our intent is to design effective and efficient interaction test software that tests the completeness of the notifications sent out by the Subject to the Observer, the correctness of the queries sent by the Observers, and the consistency of the data in the Subject and in the Observers.

An ObserverTest object will "observe" the Subject and the Observer to determine whether each test case passes or fails. The ObserverTest pattern is an extension of the PACT generative pattern language [5]. There is a need to test the interactions between the Subject and Observer objects. Both the Subject and Observer implementations have been tested in isolation; however, the interaction between the two has not been tested and can be fairly complex. The Subject may not send notifications to its Observers when it should. The Observer may not send the correct queries of the Subject.

The tests will be conducted within the context of an application whose design is the composition of the Observer pattern and several other design patterns. Some of the Subject and Observer objects may play roles in multiple design patterns. This implies that the ObserverTest pattern may only engage a portion of the interface of each class. The test code does not require any modification of either the Subject or the Observers and can focus on the behaviors and state of the objects that are specific to the Observer pattern. There are several forces that constrain the design of the test software and the specification of the ObserverTest class.

Modifying the production software to test it requires additional tests after the production software is returned to its original form. Therefore, we prefer not to modify the software.

The ObserverTest object must know when the Subject object sends out notifications. The most efficient way to achieve this is to use the test object as an Observer object. Adding the ObserverTest object as an additional Observer to the Subject does not modify the behavior of the production software since it is supposed to be designed to accept an indeterminate number of Observers.

Having a single test object that manages, accesses, and verifies the state of both the Subject and Observer simplifies the need to verify that the state of the Subject object is properly set.

The structure of this pattern is shown in Figure 5. I will Build a Test Class for each Significant Application Class as defined in the PACT pattern language [5]. Define an ObserverTest class for each Subject/Observer pair of classes. Create an ObserverTest object for each ConcreteSubject/ ConcreteObserver association. Register each test object with the ConcreteSubject object as an Observer. Each test object contains a reference to an Observer object. The ObserverTest object invokes the interaction between objects and accesses the state of both objects to determine whether the state of each is correct. The test software will Validate the Test Results Within the Test Case [5] whenever possible.

Include the interface of the Observer interface in the interface of the ObserverTest class. Inherit from the PACT abstract classes. Specify the test suite, test script and test case methods needed to test the interaction between the Subject and Observer. Select test cases based on specific types of test cases.

At the most abstract level there is only one type of test case. In the test case, an ObserverTest object invokes a Subject method that is intended to invoke a state change and a notification. When the

ObserverTest object receives notification of the change from the Subject it will check that (1) the appropriate notification is being sent and (2) its associated Observer object has completed the intended action and is now consistent with the Subject. This type of test case produces an actual test case for each state change that is intended to lead to a notification.

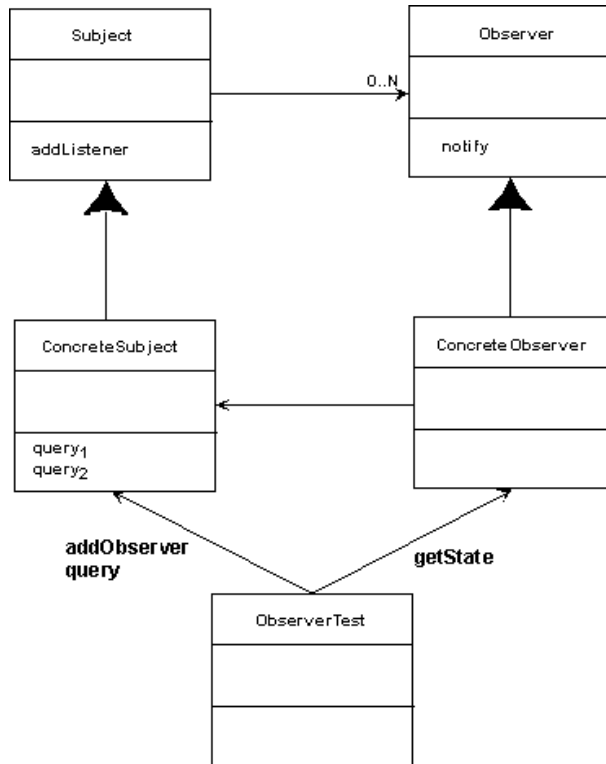


Figure 5 – The ObserverTest Pattern

The ObserverTest object uses the standard interfaces of the Subject for their interaction. Use the addObserver interface to setup the test environment. Specialize the ObserverTest class to each Observer/Subject pair so that it knows which methods in the query interface of the Subject object to use. The specialization is required to represent the unique portion of the state of the Subject that would be requested by the Observer. The ObserverTest object must know the states in which the notification should place the Observer and what messages it was designed to send to the Subject.

The ObserverTest object initiates the test sequences in response to one of its test suites being invoked. It creates the Subject and Observer objects, sets up the dependency between specific Subject/Observer objects and then invokes methods on the interface of the Subject in order to induce a notification. The sequence of events is shown in Figure 6.

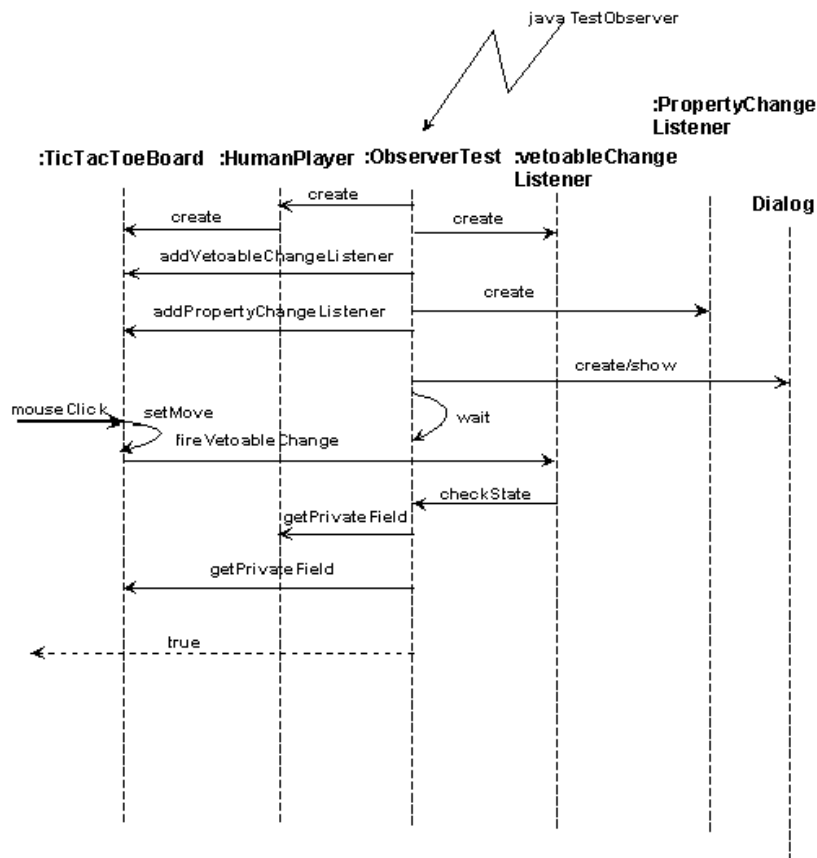


Figure 6 – ObserverTest implementation

The main consequence of applying the ObserverTest pattern is the ability to achieve an "all notifications" level of test coverage. That is, every possible notification is sent in all possible situations. This increases our confidence that the interactions have been properly designed and implemented. Since the Observer pattern is an extensibility device, we also have reason to believe the system is extensible.

The ObserverTest pattern is related to the PACT pattern language. The ObserverTest class is implemented as a PACT class. That implies it encapsulates the test cases needed to test the interactions between the Subject and Observer as well as the logic needed to verify the state of the Subject and Observer objects. As an example, consider applying the canonical test case for the Observer test pattern to code that follows the VetoableChange pattern. The TestObserver adds a VetoableChangeListener for itself with the Subject object. The TestObserver adds a VetoableChangeListener for the Observer object to the Subject. The TestObserver also adds corresponding PropertyChangeListeners. When the TestObserver receives the notification, it checks the state of the Subject and Observer objects. The two objects should be consistent, but in the original state. After the PropertyChange signal is fired, the Subject and Observer objects should be consistent, but in the new state.

Since the VetoableChange pattern is more specific than the Observer pattern, additional test cases can be defined. In particular, a second type of test case can be defined which fires a VetoableChange but results in a PropertyChangeException from the Observer rather than a PropertyChange from the Subject, shown in Figure 6. In this case, the two objects should still be consistent and should still contain the values that were present when the VetoableChange signal was fired. Several actual test cases can be constructed from this type of test case.

Evaluation of the ObserverTest Pattern

Jim Coplien evaluates a pattern on his web page [3] using the criteria that I will use here to evaluate the ObserverTest pattern.

This pattern solves a real problem. Interactions are a source of many errors in component-based software design. Writing effective test software that examines the interactions is difficult. The ObserverTest pattern provides an effective technique for constructing these tests.

The design is a proven concept because it is a variant of the Observer pattern. The test software needs to be notified when an action has happened. The ObserverTest pattern allows the software using the Observer pattern to be tested without being modified.

This solution wasn't obvious (at least to me). I have seen other approaches to this problem used in the field. However, after careful analysis of the forces, this pattern is the optimal solution.

The design describes a relationship between the application software and the test software. The pattern

describes a set of interactions. The test software initiates an interaction, the Subject object treats the ObserverTest object like any other observer, and the ObserverTest object then accesses attributes of the Subject.

The pattern has a significant human component in that it appeals to the inherent search for symmetry. (Incidentally, a variation of the ObserverTest pattern uses multiple observers, one in each process.)

Summary

Analysis of the Product Under Test is the most time-consuming, and therefore most costly, part of testing. Recognizing patterns is an important step in analysis and potentially a major time saver. By reading the design documentation and identifying standard design patterns used by the development team, the tester can apply their own patterns. The development of a test pattern requires the same level of abstraction as is found in the design pattern. The test pattern addresses the fundamental nature of the design pattern while permitting many different implementations. The code, which tests an application of the Observer pattern, will vary greatly from one instance to another. I have shown only one of a number of test patterns that many other testers, and I have used for some time. In future columns I will provide additional patterns and I welcome contributions from others in the testing community.

source: John D. McGregor - <http://www.cs.clemson.edu/~johnmc/joop/col18/column18.html>

Unit Test Design Patterns

Collection Management Patterns

A lot of what applications do is manage collections of information. While there are a variety of collections available to the programmer, it is important to verify (and thus document) that the code is using the correct collection. This affects ordering and constraints.

The Collection-Order Pattern



This is a simple pattern that verifies the expected results when given an unordered list. The test validates that the result is as expected:

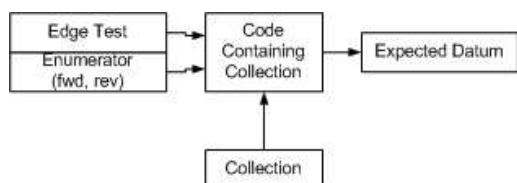
```
unordered
```

```
ordered
```

```
same sequence as input
```

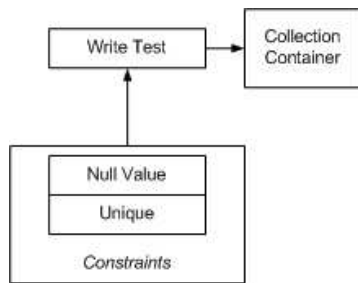
This provides the implementer with crucial information as to how the container is expected to manage the collection.

The Enumeration Pattern



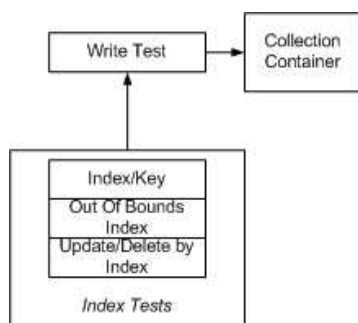
This pattern verifies issues of enumeration, or collection traversal. For example, a collection may need to be traversed forwards and backwards. This is an important test to perform when collections are non-linear, for example a collection of tree nodes. Edge conditions are also important to test--what happens when the collection is enumerated past the first or last item in the collection?

The Collection-Constraint Pattern



This pattern verifies that the container handles constraint violations: null values and inserting duplicate keys. This pattern typically applies only to key-value pair collections.

The Collection-Indexing Pattern



The indexing tests verify and document the indexing methods that the collection container must support--by index and/or by key. In addition, they verify that update and delete transactions that utilize indexing are working properly and are protected against missing indexes.

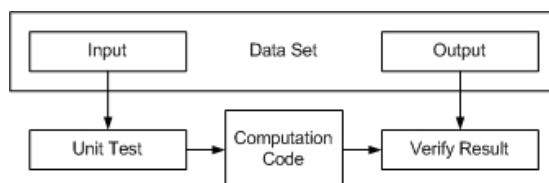
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Data Driven Test Patterns

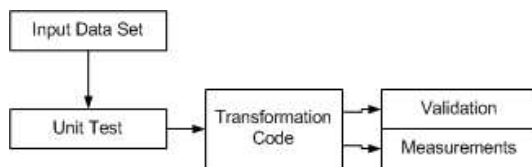
Constructing Parameter-Range unit tests is doable for certain kinds of testing, but it becomes inefficient and complicated to test a piece of code with a complex set of permutations generated by the unit test itself. The data driven test patterns reduce this complexity by separating the test data from the test. The test data can now be generated (which in itself might be a time consuming task) and modified independent of the test.

The Simple-Test-Data Pattern



In the simplest case, a set of test data is iterated through to test the code and a straightforward result (either pass or fail) is expected. Computing the result can be done in the unit test itself or can be supplied with the data set. Variances in the result are not permitted. Examples of this kind of Simple-Test-Data pattern include checksum calculations, mathematical algorithms, and simple business math calculations. More complex examples include encryption algorithms and lossless encoding or compression algorithms.

The Data-Transformation-Test Pattern



The Data-Transformation-Test pattern works with data in which a qualitative measure of the result must be performed. This is typically applied to transformation algorithms such as lossy compression. In this case, for example, the unit test might want to measure the performance of the algorithm with regard to the compression rate vs. the data loss. The unit test may also need to verify that the data can be translated back into something that resembles the input data within some tolerance. There are other applications for this kind of unit test--a rounding algorithm that favors the merchant rather than the customer is a simple example. Another example is precision. Precision occurs frequently in business--the calculation of taxes, interesting, percentages, etc., all of which ultimately must be rounded to the penny or dollar but can have dramatic effects on the resulting value if precision is not managed correctly throughout the calculation.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Data Transaction Patterns

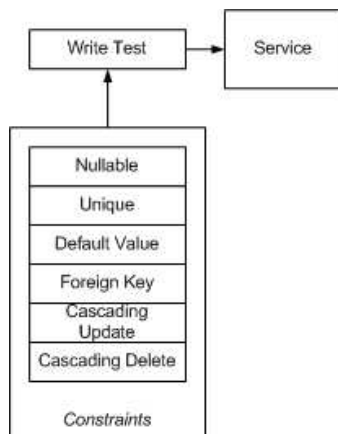
Data transaction patterns are a start at embracing the issues of data persistence and communication. More on this topic is discussed under "Simulation Patterns". Also, these patterns intentionally omit stress testing, for example, loading on the server. This will be discussed under "Stress-Test Patterns".

The Simple-Data-I/O Pattern



This is a simple data transaction pattern, doing little more than verifying the read/write functions of the service. It may be coupled with the Simple-Test-Data pattern so that a set of data can be handed to the service and read back, making the transaction tests a little bit more robust.

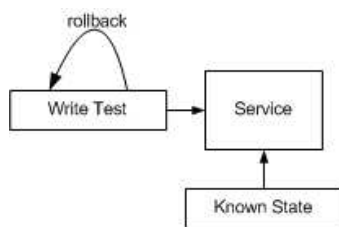
The Constraint-Data Pattern



The Constraint-Data pattern adds robustness to the Simple-Data-I/O pattern by testing more aspects of the service and any rules that the service may incorporate. Constraints typically include: can be null must be unique default value foreign key relationship cascade on update

cascade on delete As the diagram illustrates, these constraints are modeled after those typically found in a database service and are "write" oriented. This unit test is really oriented in verifying the service implementation itself, whether a DB schema, web service, or other model that uses constraints to improve the integrity of the data.

The Rollback Pattern



The rollback pattern is an adjunct to the other transaction testing patterns. While unit tests are supposed to be executed without regard to order, this poses a problem when working with a database or other persistent storage service. One unit test may alter the dataset causing another unit test to inappropriately fail. Most transactional unit tests should incorporate the ability to rollback the dataset to a known state. This may also require setting the dataset into a known state at the beginning of the unit test. For performance reasons, it is probably better to configure the dataset to a known state at the beginning of the test suite rather than in each test and use the service's rollback function to restore that state for each test (assuming the service provides rollback capability).

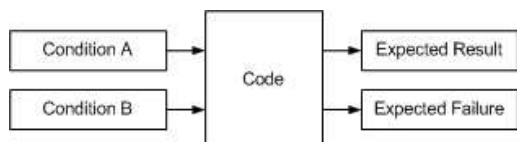
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Pass/Fail Patterns

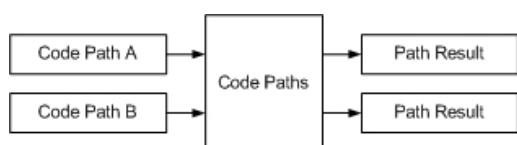
These patterns are your first line of defense (or attack, depending on your perspective) to guarantee good code. But be warned, they are deceptive in what they tell you about the code.

The Simple-Test Pattern



Pass/fail unit tests are the simplest pattern and the pattern that most concerns me regarding the effectiveness of a unit test. When a unit test passes a simple test, all it does is tell me that the code under test will work if I give it exactly the same input as the unit test. A unit test that exercises an error trap is similar--it only tells me that, given the same condition as the unit test, the code will correctly trap the error. In both cases, I have no confidence that the code will work correctly with any other set of conditions, nor that it will correctly trap errors under any other error conditions. This really just basic logic. However, on these grounds you can hear a lot of people shouting "it passed!" as all the nodes on the unit test tree turn green.

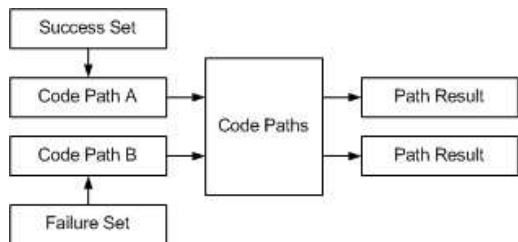
The Code-Path Pattern



The Simple-Test pattern typifies what I call "black box testing". Without inspecting the code, that's about all you can do--write educated guesses as to what the code under test might

encounter, both as success cases and failure cases, and test for those guesses. A better test ensures that at least all the code paths are exercised. This is part of "white box testing"--knowing the inside workings of the code being tested. Here the priority is not to set up the conditions to test for pass/fail, but rather to set up conditions that test the code paths. The results are then compared to the expected output for the given code path. But now we have a problem--how can you do white box testing (testing the code paths) when the code hasn't been written? Here we are immediately faced with the "design before you code" edge of that sword. The discipline here, and the benefit of unit testing by enforcing some up front design, is that the unit test can test for code paths that the implementer may not typically consider. Furthermore, the unit test documents precisely what the code path is expected to do. Conversely, discipline is needed during implementation when it is discovered that there are code paths that the unit test did not foresee--time to fix the unit test!

The Parameter-Range Pattern



Still, the above test, while improving on the Simple-Test pattern, does nothing to convince me that the code handles a variety of pass/fail conditions. In order to do this, the code should really be tested using a range of conditions. The Parameter-Range pattern does this by feeding the Code-Path pattern with more than a single parameter set. Now I am finally beginning to have confidence that the code under test can actually work in a variety of environments and conditions.

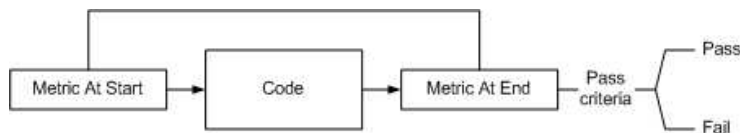
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Performance Patterns

Unit testing should not just be concerned with function but also with form. How efficiently does the code under test perform its function? How fast? How much memory does it use? Does it trade off data insertion for data retrieval effectively? Does it free up resources correctly? These are all things that are under the purview of unit testing. By including performance patterns in the unit test, the implementer has a goal to reach, which results in better code, a better application, and a happier customer.

The Performance-Test Pattern



The basic types of performance that can be measured are:

- Memory usage (physical, cache, virtual)
- Resource (handle) utilization
- Disk utilization (physical, cache)
- Algorithm Performance (insertion, retrieval, indexing, and operation)

Note that some languages and operating systems make this information difficult to retrieve. For example, the C# language with its garbage collection is rather difficult to work with in regards to measuring memory utilization. Also, in order to achieve meaningful metrics, this pattern must

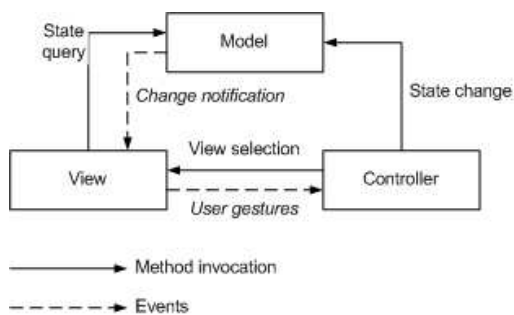
often be used in conjunction with the Simple-Test-Data pattern so that the metric can measure an entire dataset. Note that just-in-time compilation makes performance measurements difficult, as do environments that are naturally unstable, most notably networks. I discuss the issue of performance and memory instrumentation in my fourth article in a series on advanced unit testing found at <http://www.codeproject.com/csharp/autp4.asp>.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Presentation Layer Patterns

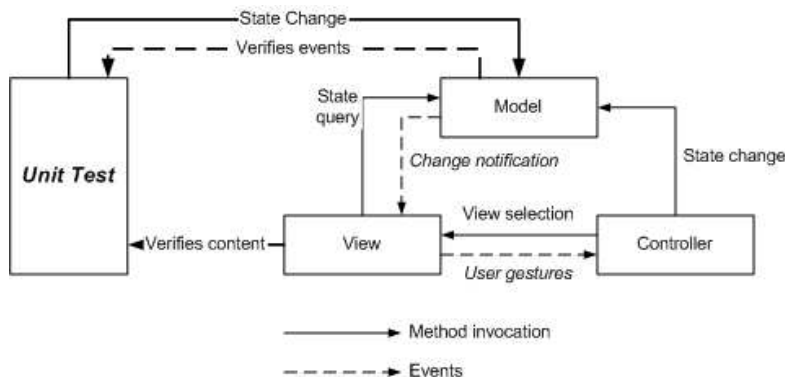
One of the most challenging aspects of unit testing is verifying that information is getting to the user right at the presentation layer itself and that the internal workings of the application are correctly setting presentation layer state. Often, presentation layers are entangled with business objects, data objects, and control logic. If you're planning on unit testing the presentation layer, you have to realize that a clean separation of concerns is mandatory. Part of the solution involves developing an appropriate Model-View-Controller (MVC) architecture. The MVC architecture provides a means to develop good design practices when working with the presentation layer. However, it is easily abused. A certain amount of discipline is required to ensure that you are, in fact, implementing the MVC architecture correctly, rather than just in word alone. Sun Microsystems has a webpage which I consider is the gospel for the MVC architecture: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>. To summarize the MVC pattern here:



It is vital that events are used for model notification changes and user gestures, such as clicking on a button. If you don't use events, the model breaks because you can't easily exchange the view to adjust the presentation to the particular presentation layer requirement. Furthermore, without events, objects become entangled. Also, events, such as managed by an event pool, allow for instrumentation and ease debugging. The only exception to this model that I have found in practice is that on occasion, a state change in the model might be captured by an event in the controller rather than the view. Some model changes, such as user authorization, are view-less but end up affecting other aspects of the controller.

The View-State Test Pattern

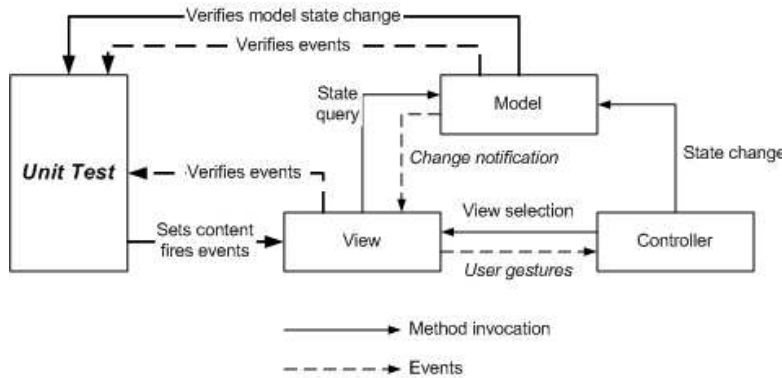
This test verifies that for a change in the model state, the view changes state appropriately.



This test exercises only half of the MVC pattern--the model event notifications to the view, and the view management of those events in terms of affects on the presentation layer content and state. The controller is not a factor in this test pattern.

The Model-State Test Pattern

Once we have verified application's performance with the View-State Pattern, we can progress to a more complicated one. In this pattern, the unit test simulates user gestures by setting state and content directly in the presentation layer, and if necessary, invoking a state change event such as "KeyUp", "Click", etc.



As diagrammed above, the unit test verifies that the model state has changed appropriately and that the expected events fired. In addition, the view's events can be hooked and verified to fire correctly for the simulated user gesture. This test may require some setup on the model itself, and treats the controller as a black box, however model state can be inspected to determine whether the controller is managing the model state appropriately.

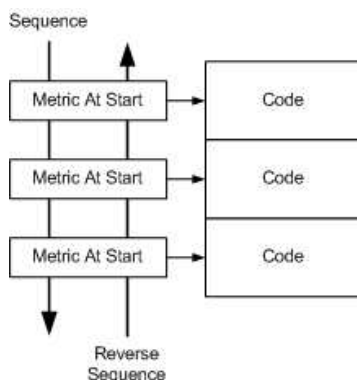
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Process Patterns

Unit testing is intended to test, well, units...the basic functions of the application. It can be argued that testing processes should be relegated to the acceptance test procedures, however I don't buy into this argument. A process is just a different type of unit. Testing processes with a unit tester provide the same advantages as other unit testing--it documents the way the process is intended to work and the unit tester can aid the implementer by also testing the process out of sequence, rapidly identifying potential user interface issues as well. The term "process" also includes state transitions and business rules, both of which must be validated.

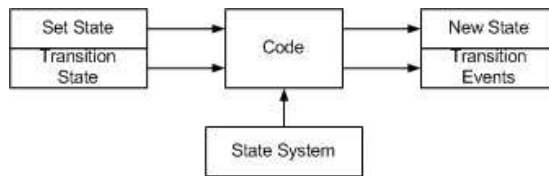
The Process-Sequence Pattern



This pattern verifies the expected behavior when the code is performed in sequence, and it

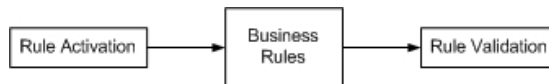
validates that problems when code is executed out of sequence are properly trapped. The Process-Sequence pattern also applies to the Data-Transaction pattern--rather than forcing a rollback, resetting the dataset, or loading in a completely new dataset, a process can build on the work of the previous step, improving performance and maintainability of the unit test structure.

The Process-State Pattern



The concept of state cannot be decoupled from that of process. The whole point of managing state is so that the process can transition smoothly from one state to another, performing any desired activity. Especially in "stateless" systems such as web applications, the concept of state (as in the state of the session) is important to test. To accomplish this without a complicated client-server setup and manual actions requires a unit tester that can understand states and allowable transitions and possibly also work with mock objects to simulate complicated client-server environments.

The Process-Rule Pattern



This test is similar to the Code-Path pattern--the intention is to verify each business rule in the system. To implement such a test, business rules really need to be properly decoupled from surrounding code--they cannot be embedded in the presentation or data access layers. As I state elsewhere, this is simply good coding, but I'm constantly amazed at how much code I come across that violates these simple guidelines, resulting in code that is very difficult to test in discrete units. Note that here is another benefit of unit testing--it enforces a high level of modularity and decoupling.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

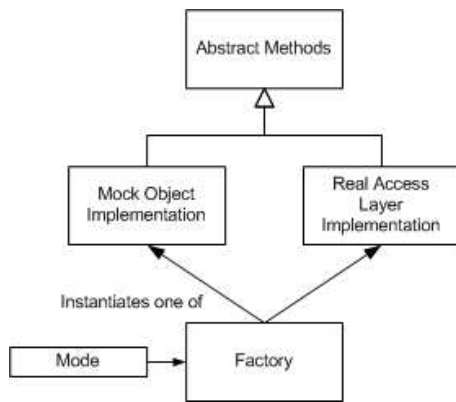
Unit Test Design Patterns

Simulation Patterns

Data transactions are difficult to test because they often require a preset configuration, an open connection, and/or an online device (to name a few). Mock objects can come to the rescue by simulating the database, web service, user event, connection, and/or hardware with which the code is transacting. Mock objects also have the ability to create failure conditions that are very difficult to reproduce in the real world--a lossy connection, a slow server, a failed network hub, etc. However, to properly use mock objects the code must make use of certain factory patterns to instantiate the correct instance--either the real thing or the simulation. All too often I have seen code that creates a database connection and fires off an SQL statement to a database, all embedded in the presentation or business layer! This kind of code makes it impossible to simulate without all the supporting systems--a preconfigured database, a database server, a connection to the database, etc. Furthermore, testing the result of the data transaction requires another transaction, creating another failure point. As much as possible, a unit test should not in itself be subject to failures outside of the code it is trying to test.

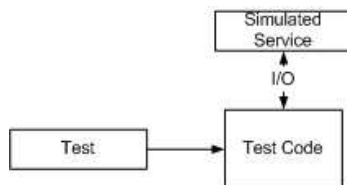
Mock-Object Pattern

In order to properly use mock objects, a factory pattern must be used to instantiate the service connection, and a base class must be used so that all interactions with the service can be managed using virtual methods. (Yes, alternatively, Aspect Oriented Programming practices can be used to establish a pointcut, but AOP is not available in many languages). The basic model is this:



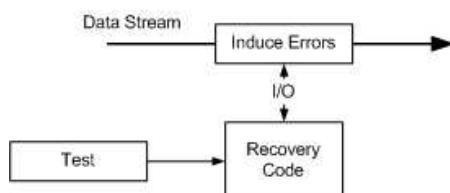
To achieve this construct, a certain amount of foresight and discipline is needed in the coding process. Classes need to be abstracted, objects must be constructed in factories rather than directly instantiated in code, facades and bridges need to be used to support abstraction, and data transactions need to be extracted from the presentation and business layers. These are good programming practices to begin with and result in a more flexible and modular implementation. The flexibility to simulate and test complicated transactions and failure conditions gains a further advantage to the programmer when mock objects are used.

The Service-Simulation Pattern



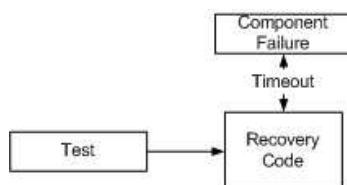
This test simulates the connection and I/O methods of a service. In addition to simulating an existing service, this pattern is useful when developing large applications in which functional pieces are yet to be implemented.

The Bit-Error-Simulation Pattern



I have only used this pattern in limited applications such as simulating bit errors induced by rain-fade in satellite communications. However, it is important to at least consider where errors are going to be handled in the data stream--are they handled by the transport layer or by higher level code? If you're writing a transport layer, then this is a very relevant test pattern.

The Component-Simulation Pattern



In this pattern, the mock object simulates a component failure, such as a network cable, hub, or other device. After a suitable time, the mock object can do a variety of things:

throw an exception

```
return incomplete or completely missing data
return a "timeout" error
```

Again, this unit test documents that the code under test needs to handle these conditions.

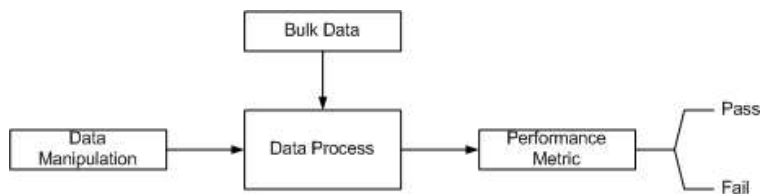
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Stress-Test Patterns

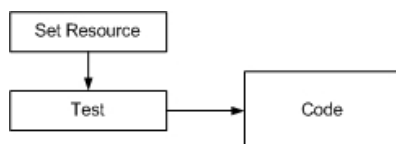
Most applications are tested in ideal environments--the programmer is using a fast machine with little network traffic, using small datasets. The real world is very different. Before something completely breaks, the application may suffer degradation and respond poorly or with errors to the user. Unit tests that verify the code's performance under stress should be met with equal fervor (if not more) than unit tests in an ideal environment.

The Bulk-Data-Stress-Test Pattern



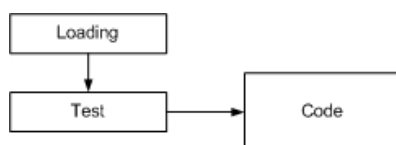
This test is designed to validate the performance of data manipulation when working with large data sets. These tests will often reveal inefficiencies in insertion, access, and deletion processes which are typically corrected by reviewing the indexing, constraints, and structure of the data model, including whether code is should be run on the client or the server.

The Resource-Stress-Test Pattern



Resource consumption stress testing depends on features of the operating system and may be served better by using mock objects. If the operating system supports simulating low memory, low disk space, and other resources, then a simple test can be performed. Otherwise, mock objects must be used to simulate the response of the operating system under a low resource condition.

The Loading-Test Pattern



This test measures the behavior of the code when another machine, application, or thread is loading the "system", for example high CPU usage or network traffic. This is a simulation only (which does not use mock objects) and therefore is of dubious value. Ideally, a unit test that is intended to simulate a high volume of network traffic would create a thread to do just that--inject packets onto the network.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Test Harness Design Patterns

The Microsoft® .NET Framework provides you with many ways to write software test automation. But in conversations with my colleagues I discovered that most engineers tend to use only one or two of the many fundamental test harness design patterns available to them. Most often this is true because many developers and testers simply aren't aware that there are more possibilities.

Furthermore I discovered that there is some confusion and debate about when to use a lightweight test harness and when to use a more sophisticated test framework like the popular NUnit. In this month's column James Newkirk, the original author of NUnit, joins me to explain and demonstrate how to use fundamental lightweight test harness patterns and also show you their relation to the more powerful NUnit test framework. The best way to show you where the two of us are headed is with three screen shots. Suppose you are developing a .NET-based application for Windows®. The screen shot in Figure 1 shows the fairly simplistic but representative example of a poker game. The poker application references a PokerLib.dll library that has classes to create and manipulate various poker objects. In particular there is a Hand constructor that accepts a string argument like "Ah Kh Qh Jh Th" (ace of hearts through 10 of hearts) and a Hand.GetHandType method that returns an enumerated type with a string representation like RoyalFlush.



Figure 1 System Under Test

Now suppose you want to test the underlying PokerLib.dll methods for functional correctness. Manually testing the library would be time-consuming, inefficient, error-prone, and tedious. You have two better testing strategies. A first alternative to manual testing is to write a lightweight test harness that reads test case input and expected values from external storage, calls methods in the library, and compares the actual result with the expected result. When using this approach, you can employ one of several basic design patterns. Figure 2 shows a screen shot of a test run that uses the simplest of the design patterns. Notice that there are five test cases included in this run; four cases passed and one failed. The second alternative to manual testing is to use a test framework. Figure 3 shows a screen shot of a test run which uses the NUnit framework.

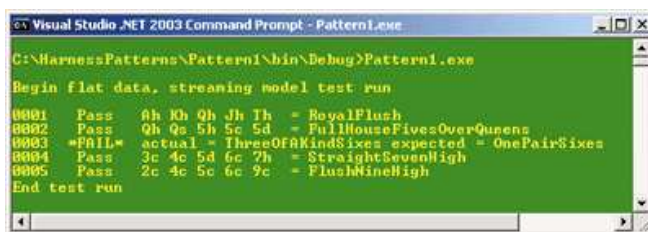


Figure 2 Lightweight Test Harness Run

In the sections that follow, we will explain fundamental lightweight test harness design patterns, show you a basic NUnit test framework approach, give you guidance on when each technique is most appropriate, and describe how you can adapt each technique to meet your own needs. You'll learn the pros and cons of multiple test design patterns, and this information will be a valuable addition to your developer, tester, and manager skill sets.

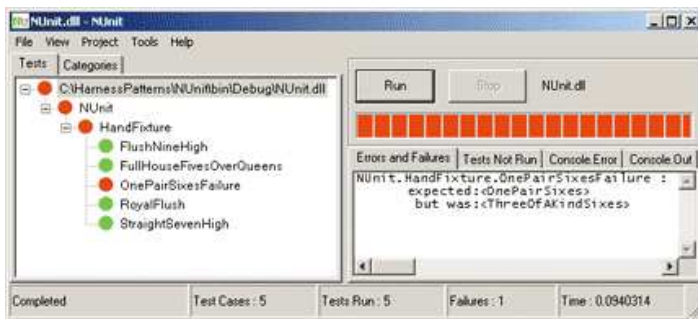


Figure 3 NUnit Test Framework Run

The Six Basic Lightweight Test Harness Patterns

It is useful to classify lightweight data-driven test harness design patterns into six categories based on type of test case storage and test case processing model. There are three fundamental types of test case storage: flat file, hierarchical, and relational. Additionally, there are two fundamental processing models: streaming and buffered. This categorization leads to six test harness design patterns, the cross-product of the storage types with the processing models.

Of course you can think of many other possibilities, but these six categories give you a practical way to think about structuring your lightweight test harnesses. Notice that this assumes that the test case storage is external to the test harness code. In general, external test case storage is better than embedding test case data with the harness code because external storage can be edited and shared more easily than embedded data. However, as we'll explain later, the test-driven approach is primarily a developer activity and typically uses embedded test case data which does have certain advantages over external data. Separately, NUnit can be used with external test case storage and can support both streaming and buffered processing models.

Flat Test Case Data

The most rudimentary type of test case data is flat data. The data in Figure 4 is the test case file used to generate the test run shown in Figure 2. Compared with hierarchical data and relational data, flat data is most appropriate when you have simple test case input and expected values, you are not in an XML-dominated environment, and you do not have a large test management structure.

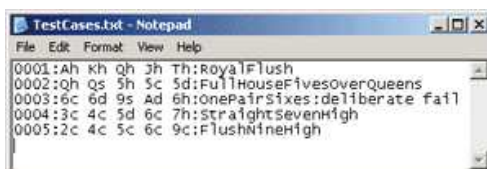


Figure 4 Flat Test Case Data

At a minimum every test case has an ID, one or more inputs, and one or more expected results. There is nothing profound about how to store test case data. Examples of flat data are text files, Excel worksheets, and individual tables in a database. Examples of hierarchical data stores are XML files and some .ini files. SQL Server™ databases and Access databases are examples of relational data stores when multiple tables are used in conjunction through relationships. Here you can see we're using a simple text file with a test case ID field, a single input field, and a single expected result field—simple and effective. We will discuss the pros and cons of each of the three storage types later in this column.

This pseudocode shows the basic streaming processing model:

```

open test case data store
loop
  read a test case
  parse id, input, expected
  send input to system under test
  if actual result == expected result
    write pass result to external results file
  else
    write fail result to external results file
  end if
end loop

```

The code in Figure 5 shows the main loop. The algorithm is implemented in Visual Basic® .NET, but any .NET-targeted language could be used. The complete source code for all examples is available in the code download that accompanies this column.

```

while ((line = sr.ReadLine()) != null) // main loop
{
    tokens = line.Split(':'); // parse input
    caseid = tokens[0];
    cards = tokens[1].Split(' ');
    expected = tokens[2];

    Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]); // test
    actual = h.GetHandType().ToString();
    Console.WriteLine(caseid + " ");
    sw.WriteLine(caseid + " ");
    if (actual == expected) // determine result
    {
        string rv = string.Format(" Pass   {0} = {1}",
            h.ToShortString(), actual);
        Console.WriteLine(rv);
        sw.WriteLine(rv);
    }
    else
    {
        string rv = string.Format(" *FAIL*  actual = {0} expected = {1}",
            actual, expected);
        Console.WriteLine(rv);
        sw.WriteLine(rv);
    }
} //main loop

```

Figure 5 Streaming Flat Data Design

Notice that we echo test results to the command shell with a `Console.WriteLine` statement and write test results to an external text file with a call to `StreamWriter.WriteLine`. In general, it makes sense to save test case results to the same type of storage as your test case data, but this is considered to be more a matter of consistency than a technical issue.

We call the algorithm a streaming model because it resembles the .NET input-output streaming model; there is a continuous stream of test case input and test results. Now let's look at the buffered model. The pseudocode in Figure 6 is what we'll call the buffered processing model.

```

open test case data store
loop
    read a test case from external storage
    save test data to in-memory data store
end loop

loop
    read a test case
    parse test case id, input, expected
    send input to system under test
    if actual result == expected result
        write pass result to in-memory data store
    else
        write fail result to in-memory data store
    end if
end loop

loop
    read test result from in-memory store
    write test result to external storage
end loop

```

Figure 6 Buffered Algorithm

With the buffered test harness model we read all test case data into memory before executing any test cases. All test results are saved to an in-memory data store and then emitted to external storage after all test cases have been executed. In other words, test case input and results are buffered through the test system rather than streamed through the system. The code snippet in Figure 7 shows you how we implemented the buffered model using the test case data file that is shown in Figure 4.

```

// 1. read test case data into memory
ArrayList cases = new ArrayList();
string line;
while ((line = sr.ReadLine()) != null) cases.Add(line);

// 2. main test processing loop
ArrayList results = new ArrayList();
string caseid, expected, actual, result;
string[] tokens, cards;
for (int i = 0; i < cases.Count; ++i)
{
    tokens = cases[i].ToString().Split(':'); // parse input
    caseid = tokens[0];
    cards = tokens[1].Split(' ');
    expected = tokens[2];

    Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
    actual = h.GetHandType().ToString();
    result = caseid + " " + (actual == expected ?
        " Pass " + h.ToShortString() + " = " + actual :
        " *FAIL* actual = " + actual + " expected = " + expected);
    results.Add(result); // store result into memory
}

// 3. emit results to external storage
for (int i = 0; i < results.Count; ++i)
{
    Console.WriteLine(results[i].ToString());
    sw.WriteLine(results[i]);
}

```

Figure 7 Flat Data Buffered Design

If you compare the streaming processing model with the buffered model, it's pretty clear that the streaming model is both simpler and shorter. So why would you ever want to use the buffered model? There are two common testing scenarios where you should consider using the buffered processing model instead of the streaming model. First, if the aspect in the system under test involves file input/output, you often want to minimize the test harness file operations. This is especially true if you are monitoring performance. Second, if you need to perform some pre-processing of your test case input or post-processing of your test case results (for example aggregating various test case category results), it's almost always more convenient to have all results in memory where you can process them. The NUnit test framework is very flexible and can use external test case storage primarily with the buffered processing models. However, a complete discussion of how to use NUnit in these ways would require an entire article by itself and is outside the scope of this column.

Hierarchical Test Case Data

Hierarchical test case data, especially XML, has become very common. In this section we will show you the streaming and buffered lightweight test harness processing models when used in conjunction with XML test case data. Compared with flat test case data and relational data, hierarchical XML-based test case data is most appropriate when you have relatively complex test case input or expected results, or you are in an XML-based environment (your development and test effort infrastructure relies heavily on XML technologies). Here is a sample of XML-based test case data that corresponds to the flat file test case data in Figure 4:

```

<?xml version="1.0" ?>
<TestCases>
  <case caseid="0001">
    <input>Ah Kh Qh Jh Th</input>
    <expected>RoyalFlush</expected>
  </case>
  <case caseid="0002">
    <input>Qh Qs 5h 5c 5d</input>
    <expected>FullHouseFivesOverQueens</expected>
  </case>
  ...
</TestCases>

```

Because XML is so flexible there are many hierarchical structures we could have chosen. For example, the same test cases could have been stored as follows:

```

<?xml version="1.0" ?>
<TestCases>
  <case caseid="0001" input="Ah Kh Qh Jh Th" expected="RoyalFlush" />
  <case caseid="0002" input="Qh Qs 5h 5c 5d"
    expected=" FullHouseFivesOverQueens" />
  ...
</TestCases>

```

Just as with flat test case data, you can use a streaming processing model or a buffered model. In each case the algorithm is the same as shown in the basic streaming processing model algorithm and in the buffered algorithm that was shown in Figure 6. Interestingly though, the XML test case data model implementations are quite different from their flat data

counterparts. Figure 8 shows key code from a C#-based streaming model implementation.

```

xtr.WriteStartElement();
xtr.WriteStartElement("TestResults");

while (!xtr.EOF) // main loop
{
    if (xtr.Name == "TestCases" && !xtr.IsStartElement()) break;

    while (xtr.Name != "case" || !xtr.IsStartElement())
        xtr.Read(); // advance to a <case> element if not there yet

    caseid = xtr.GetAttribute("caseid");
    xtr.Read(); // advance to <input>
    input = xtr.ReadElementString("input"); // advance to <expected>
    expected = xtr.ReadElementString("expected"); // advance to </case>
    xtr.Read(); // advance to next <case> or </TestCases>

    cards = input.Split(' ');
    Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
    actual = h.GetHandType().ToString();

    xtr.WriteStartElement("result");
    xtr.WriteStartAttribute("caseid", null);
    xtr.WriteString(caseid);
    xtr.WriteEndElement();
    xtr.WriteString(actual == expected ?
        " Pass " + h.ToShortString() + " = " + actual :
        " *FAIL* actual = " + actual + " expected = " + expected);
    xtr.WriteEndElement(); // </result>
} // main loop

```

Figure 8 XML Data Streaming Design

With a streaming model, we use an `XmlTextReader` object to read one XML node at a time. But because XML is hierarchical it is a bit tricky to keep track of exactly where we are within the file, especially when the nested becomes more extreme (in this particular example, the data is little more than a flat file, but it could be significantly more complex). We use an `XmlTextWriter` object to save test results in XML form. Now we'll show you a buffered approach for XML test case data. Figure 9 shows key code from a buffered processing model implementation.

```

// 1. read test case data into memory
XmlSerializer xds = new XmlSerializer(typeof(TestCases));
TestCases tc = (TestCases)xds.Deserialize(sr);

// 2. processing loop
string expected, actual;
string[] cards;
TestResults tr = new TestResults(tc.Items.Length);

for (int i = 0; i < tc.Items.Length; ++i) // test loop
{
    SingleResult res = new SingleResult();
    res.caseid = tc.Items[i].caseid;
    cards = tc.Items[i].input.Split(' ');
    expected = tc.Items[i].expected;

    Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
    actual = h.GetHandType().ToString();

    res.result = (actual == expected) ? // store results into memory
        "Pass " + h.ToShortString() + " = " + actual :
        " *FAIL* " + "actual = " + actual + " expected = " + expected;
    tr.Items[i] = res;
}

// 3. emit results to external storage
XmlTextWriter xtw = new XmlTextWriter("..\..\TestResults.xml", System.Text.Encoding.UTF8);
XmlSerializer xs = new XmlSerializer(typeof(TestResults));
xs.Serialize(xtw, tr);

```

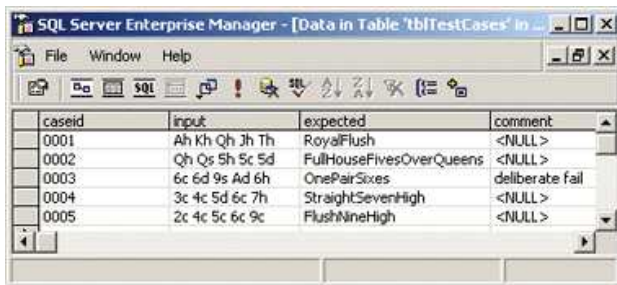
Figure 9 XML Data Buffered Design

We use an `XmlSerializer` object from the `System.Xml.Serialization` namespace to read the entire XML test case file into memory with a single line of code and also to write the entire XML result file with a single line of code. Of course, this requires us to prepare appropriate collection classes (`TestCases` and `TestResults` in the code) to hold the data. Unlike flat test case data, with XML data the buffered model test harness code tends to be shorter and simpler. So when might you consider using a streaming model in conjunction with XML test case data? Most often you will want to use a streaming model when you have a lot of test cases to deal with. Reading a huge amount of test case data into memory all at once may not always be possible, especially if you are running stress tests under conditions of reduced internal memory.

Relational Test Case Data

In this section we'll describe the streaming and buffered lightweight test harness processing models when used in conjunction with SQL test case data. Compared with flat data and hierarchical data, relational SQL-based test case data is most appropriate when you have a very large number of test cases, or you are in a relatively long product cycle (because

you will end up having to store lots of test results), or you are working in a relatively sophisticated development and test infrastructure (because you will have lots of test management tools). Figure 10 shows test case data that has been stored in a SQL database.



caseid	input	expected	comment
0001	Ah Kh Qh Jh Th	RoyalFlush	<NULL>
0002	Qh Qs 5h 5c 5d	FullHouseFivesOverQueens	<NULL>
0003	6c 6d 9s Ad 6h	OnePairSixes	deliberate fail
0004	3c 4c 5d 6c 7h	StraightSevenHigh	<NULL>
0005	2c 4c 5c 6c 9c	FlushNineHigh	<NULL>

Figure 10 SQL-based Test Case Data

Just as with flat test case data and hierarchical data, you can use a streaming processing model or a buffered model. The basic streaming and buffered models described earlier will be the same. The streaming model implementation is included in this column's download. If you examine the code you'll see that for a streaming model we like to use a `SqlDataReader` object and its `Read` method. For consistency we insert test results into a SQL table rather than save to a text file or XML file. We prefer to use two SQL connections—one to read test case data and one to insert test results. As with all the techniques in this column, there are many alternatives available to you.

The code for the buffered processing model can be downloaded from the MSDN Magazine Web site. Briefly, we connect to the test case database, fill a `DataSet` with all the test case data, iterate through each case, test, store all results into a second `DataSet`, and finally emit all results to a SQL table.

Using relational test case data in conjunction with ADO.NET provides you with many options. Assuming memory limits allow, we typically prefer to read all test case data into a `DataSet` object. Because all the test case data is in a single table, we could also have avoided the relatively expensive overhead of a `DataSet` by just using a `DataTable` object. However in situations where your test case data is contained in multiple tables, reading into a `DataSet` gives you an easy way to manipulate test case data using a `DataRelation` object. Similarly, to hold test case results we create a second `DataSet` object and a `DataTable` object. After running all the test cases we open a connection to the database that holds the results table (in this example it's the same database that holds the test case data) and write results using the `SqlDataAdapter.Update` method. Recall that when using flat test case data, a streaming processing model tends to be simpler than a buffered model, but that when using hierarchical XML data, the opposite is usually true. When using test case data stored in a single table in SQL Server, a streaming processing model tends to be simpler than a buffered model and the technique of choice. When test case data spans multiple tables, you'll likely want to use a buffered processing model.

The TDD Approach with the NUnit Framework

In the previous sections you've seen six closely related lightweight test harness design patterns. A significantly different but complementary approach is to use an existing test framework. The best known framework for use in a .NET environment is the elegant NUnit framework as shown in Figure 3. See the MSDN®Magazine article by James Newkirk and Will Stott at [Test-Driven C#: Improve the Design and Flexibility of Your Project with Extreme Programming Techniques](#) for details. The code snippet in Figure 11 shows how you can use NUnit to create a DLL that can be used by NUnit's GUI interface. And the code snippet in Figure 12 shows how you can use NUnit with external XML test case data to create a DLL that can be used by NUnit's command-line interface.

```
[Suite]
public static TestSuite Suite
{
    get
    {
        TestSuite testSuite = new TestSuite("XML Buffered Example");
        using(StreamReader reader = new StreamReader("TestCases.xml"))
        {
            XmlSerializer xds = new XmlSerializer(typeof(TestCases));
            TestCases testCases = (TestCases)xds.Deserialize(reader);

            foreach(Case testCase in testCases.cases)
            {
                string[] cards = testCase.input.Split(' ');
                HandType expectedHandType = (HandType)Enum.Parse(
                    typeof(HandType), testCase.expected);
                Hand hand = new Hand(cards[0], cards[1], cards[2],
                    cards[3], cards[4]);
                testSuite.Add(new HandTypeFixture(
                    testCase.id, expectedHandType, hand.GetHandType()));
            }
        }
        return testSuite;
    }
}
```

Figure 12 NUnit Approach with External XML Test Cases

```

using NUnit.Framework;
using PokerLib;

[TestFixture]
public class HandFixture
{
    [Test]
    public void RoyalFlush()
    {
        Hand hand = new Hand("Ah", "Kh", "Qh", "Jh", "Th");
        Assert.AreEqual(HandType.RoyalFlush, hand.GetHandType());
    }
    ... // other tests here
}

```

Figure 11 NUnit Approach with Embedded Test Cases

You may be wondering whether it's better to use NUnit or to write a custom test harness. The best answer is that it really depends on your scenarios and environment, but using both test techniques together ensures a thorough test effort. The NUnit framework and lightweight test harnesses are designed for different testing situations. NUnit was specifically designed to perform unit testing in a test-driven development (TDD) environment, and it is a very powerful tool. A lightweight test harness is useful in a wide range of situations, such as when integrated into the build process, and is more traditional than the NUnit framework in the sense that a custom harness assumes a conventional spiral-type software development process (code, test, fix).

A consequence of NUnit's TDD philosophy is that test case data is typically embedded with the code under test. Although embedded test case data cannot easily be shared (for example when you want to test across different system configurations), embedded data has the advantage of being tightly coupled with the code it's designed to test, which makes your test management process easier. Test-driven development with NUnit helps you write code and test it. This is why embedded tests with NUnit are acceptable—you change your tests as you change your code. Now this is not to say that the two test approaches are mutually exclusive; in particular NUnit works nicely in a code-first, test-later environment, and can utilize an external test case data source. And a lightweight test harness can be used in conjunction with a TDD philosophy. The NUnit framework and custom lightweight test harnesses have different strengths and weaknesses. Some of NUnit's strengths are that it is a solid, stable tool, it is a nearly a de-facto standard because of widespread use, and it has lots of features. The strengths of custom test harnesses are that they are very flexible, allowing you to use internal or external storage in a variety of environments, test for functionality as well as performance, stress, security and other types of testing, and execute sets of individual test cases or multiple state change test scenarios.

Conclusion

Let's briefly summarize. When writing a data-driven lightweight test harness in a .NET environment you can choose one of three types of external test case data storage: flat data (typically a text file), hierarchical data (typically an XML file), or relational data (typically a SQL Server database). Often you will have no choice about the type of data store to use because you will be working in an already existing development environment. Flat data is good for simple test case scenarios, hierarchical data works very well for technically complex test case scenarios, and relational data is best for large test efforts. When writing a lightweight test harness you can employ either a streaming processing model or you can choose a buffered processing model. A streaming processing model is usually simpler except when used with truly hierarchical or relational data, in which case the opposite is true. A streaming model is useful when you have a very large number of test cases, and a buffered model is most appropriate when you are testing for performance or when you need to process test cases and results. Using a test framework like NUnit is particularly powerful for unit testing when you are employing a TDD philosophy. With the .NET environment and powerful .NET-based tools like NUnit, it's possible to write great test automation quickly and efficiently. The release of Visual Studio® 2005 will only enhance your ability to write test automation and the Team System version in Visual Studio 2005 will have many NUnit-like features. With software systems increasing in complexity, testing is more important than ever. Knowledge of these test harness patterns as well as of frameworks like NUnit will help you test better and produce better software systems.

source: James McCaffrey and James Newkirk - <http://msdn.microsoft.com/en-us/magazine/cc163752.aspx>

Application Test Design Patterns

Common Test Patterns

The usual test-design process is very repetitive, yet many testing problems have generic elements that you can reuse easily in creation of later test designs. You can capture these patterns, and use them to simplify the creation of later test designs. One problem that still limits effectiveness and efficiency in the testing of applications is that a usual development-and-test cycle involves a lot of creation of test-design details from scratch, and in a somewhat ad-hoc manner, whenever a new set of application features is defined. The primary tool often is an empty test-design template, which includes categories of test issues to address enumeration of test specifications or test-case details that are based on feature functional specifications.

If viewed across a series of project cycles, test-design activities are one area in which there is a lot of repetition: Test engineers start with a blank page in a test-design specification and fill it in detail by detail, when many of these issues are generic and common to nearly every application-testing scenario.

A goal of the test-design process should be to reduce ad-hoc reinvention of test designs by capturing and reusing common test patterns.

The recognition of test patterns is loosely similar to design patterns that are used in software development. (There are also some classic object-oriented design patterns that might be suitable for test-automation design, but we do not address those patterns directly.) Creating test designs by reusing patterns can cover repetitive, common core verification of many aspects of an application. This frees the test engineer to focus more time and thought on truly unique aspects that are particular to an application without having to waste effort on definition of the more generic portions of the test design.

Just as developers who are writing software strive to minimize the code that they reinvent or create from scratch by using existing functionality for common, generic, or repeated operations, so too should test engineers strive to minimize the amount of time they spend on defining test cases and verification steps for functionality that is common to every application with which they are presented. Test engineers should be able to reuse the full and rich set of tests that previously have been identified for similar situations.

Expert-Tester Pattern

Experienced, expert testers often approach these issues by applying their knowledge and past experience as heuristics to recognize what test situations align with techniques, test issues, and test approaches. When they see a given test situation, they immediately identify a method of testing that is appropriate and applicable. Novices, on the other hand, are likely to be unaware of all of the issues or pitfalls that are associated with the situation or the test conditions that should be applied. In addition, expert knowledge is not captured explicitly into a highly transferrable format for the novice to be able to take advantage of it, and it often can be lost to the team when the expert has moved on.

Test-Category Checklists

An underused tool that provides the first step in capturing product-testing experience is the checklist. Checklists make the test process and accumulated ideas explicit for reuse at a later time by the same or another test engineer. Often, they are lists of heuristics and mnemonic devices that should be applied to the testing and verification process, to avoid forgetting any obvious tests. Effective checklists in this context are abstracted so that they are applicable to multiple projects or applications. The appropriate level of abstraction is to: Check each entry field that has maximum-length string data.

- Check each required field that has null data.
- Challenge type-checking validation of each field that has data of a different type from what is allowed.
- Test for all data values that are simultaneously maximized or most stressful.
- Verify that the nominal value of data is accepted, stored, retrieved, and displayed.

Unfortunately, generic and feature-specific test objectives often are jumbled together, without thought being given to reuse; the generic aspects of the design are buried in the specifics, and the value for reuse is reduced.

Failure Modes

Successful tests are ones that find problems; the problems that are found often are due to timeless mistakes and common failure modes. Some problem areas have classic errors in implementation; some conceptual designs have classic mistakes that are associated with them; some technologies have inherent flaws or limitations; some failures are critical or important, and must be validated; and some error-handling consistently is neglected.

The test-design processes of expert testers often rely, in part, on experience about things that have gone wrong in the past. However, this past experience is not included in any functional specification for a product, and therefore is not called out explicitly to verify. The experience of an expert tester can be captured and turned into valuable test-design heuristics.

Testing is organized often around failure modes, so that enumeration of failure modes is a valuable portion of test design. Lack of anticipation of new, novel, or infrequent failure modes is the prime source of test-escape problems, where bugs are not detected during testing phases, but affect customers who use the application. Often, these bugs are the result of failure modes that could have been anticipated—based upon past product-testing experience—but the experience was not captured, shared, or passed to a novice tester. An easy way to avoid this problem is to write generic specifications for common failure modes, convert them into test conditions to verify or validate, and concatenate this to any test-design template.

Enumeration of Things to "Cover"

A useful approach to the multidimensional, orthogonal nature of test planning is to identify categories of tests that have strong conceptual relationships, and then use those subsets to ensure that testing completeness is achieved in each category. One concept that is helpful is to break the problem into distinct manageable categories, based on the idea of "coverage." In this case, the concept of coverage can be applied in a more generic, broadly applicable form than the specific techniques that are used to identify untested areas of the code (such as the metrics that are collected with code-coverage toolsets).

The key part of this analysis process is to create as many separate categories of things that can be "covered" as possible. Iterate each category, to build lists of test items that should be covered. Common categories to use as a starting point for the process are user-scenario coverage, specification-requirement coverage, feature coverage, UI control coverage, field coverage, form coverage, output/report coverage, error coverage, code coverage, condition coverage, and event coverage.

The concept can be extended to many other areas. Identification of a number of test categories aids the analysis and design process that often is applied to requirements review for test identification. Having a formalized set of categories allows the test engineer to decompose requirements into testable form, by dividing the problem into testable fragments or test matrices across a number of dimensions.

Generic Test-Design Templates

Test-design specification templates should be expanded beyond the typical empty-document sections to fill in, to include general validation rules, test-case checklists, common failure modes, and libraries of generic tests. The templates should include many of the usual tests for application features, so that they do not have to be added or reinvented. Part of this generic design should implement "boilerplate" tests that will be the first ones that every testing effort should include, even before considering more sophisticated and instance-specific tests. In this way, the generic test-design specifications allow any tester—for example, a new contingent staff member—to perform a range of generic tests, even before any customized, application-specific test planning.

Pesticide Paradox

Test automation—especially, when it is applied (as usual default) as regression test automation—suffers from static test data. Without variation in the data or execution path, bugs are located with decreasing frequency. The paradox is that the reduction in bugs that are found per test pass is not representative of the total number of bugs that remain in the system. Bugs are only less common on the well-exercised code paths that are represented by the static data. Bugs that remain in untested areas of the application are resistant to the testing methods (the automation pesticide paradox).

The lack of variation sometimes can improve the stability of the test code somewhat, but at the detriment of good test coverage. Effectiveness of testing—as measured in the detection of new bugs—is improved significantly by the addition of variation, even a test pass using different values of data within the same equivalence class. Automation should be designed so that a test-case value can provide a common equivalence class or data type as a parameter, and the automation framework will vary the data randomly within that class and apply it for each test run. (The automation framework should record the seed values or actual data that is used, to allow reruns and retesting with the same data for debugging purposes.)

In some cases, working with dynamically selected input data can be problematic because determining the expected outcome may be difficult (the Oracle problem). However, in many cases, there are tests that can be selected and verified automatically, even without requiring sophisticated approaches to determine the correct test output for a particular input.

Test-automation strategies too often are based solely on the automation of regression tests, putting all of the eggs in that basket alone. This is analogous to buying insurance to protect against failures that are due to regression: It is expensive, and it mitigates failures in the areas that the insurance (automation) covers, but it does not reduce or prevent the problem in uninsured (unautomated) portions of the application. Wise insurance-purchasing strategies usually are based on the selection of the optimal amount, and not on the spending of all available resources on a gold-plated, zero-deductible insurance policy. In testing terms, some teams spend too much time on automation—at the expense of lost mindshare that could be dedicated to other types of testing, and putting their projects at increased risk of detecting only a small portion of the real problems.

The cost/benefit ratio of test automation can be tilted in a positive direction by concentrating on adding stability to development processes, identifying sources of regression failures, and fixing them systematically. When pursued with vigor, it soon becomes inappropriate to over-invest in a regression test-automation strategy, and much more important to invest in other more leveraged forms of test automation. Otherwise, the dominant problem changes quickly from creating tests to find bugs to trying to manage and maintain myriad redundant, overlapping, obsolete, or incorrect regression test cases.

A regression-focused approach can result in automated tests that are too static. The cost/benefit ratio is marginal for these tests when all of the maintenance and management costs are taken into account; they do not find many bugs. If test systems are designed to challenge dynamically the full range of data-dependent failures, alternate invocation modes, and number iteration they can accomplish much more than regression testing.

By increasing the abstraction of test automation to "find all strings on the form and test them with all generic tests (apply random data variations until an error occurs or 100,000 conditions are tried)" and running the tests continually, testing is elevated from marginally valuable static regression suite to a generic tool that has a much higher probability of adding value to the project. (It also has the added benefit of providing technical opportunities for extending Software Test Engineers, instead of just chaining their desk to maintain legacy automation.)

The other benefit of higher-level abstraction and making tests more generic is efficiency; adding test cases should be simple and

quick, and not require hours of custom test-case coding of minutiae. Having test automation designed at the level of "test this control()"—instead of 50 lines of "click here, type this, wait for this window"—empowers a broad range of testers to automat tests and focus efforts on planning meaningful tests, instead of having to slog through the implementation details of simplistic scenarios that are coded against poorly abstracted test frameworks.

Conclusion

Testers should build test designs around reusable test patterns that are common to a large number of application-testing problems. The inclusion of these patterns as a standard part of the test-design template reduces the time that is spent on test design, by eliminating the need to start with a blank test-design specification. Reuse of these patterns has the added benefit of codifying expert-tester knowledge, so as to increase the likelihood of catching common failures.

source: Mark Folkerts, Tim Lamey, and John Evans - <http://msdn.microsoft.com/en-us/testing/cc514239>.

Application Test Design Patterns

Reuse Test Patterns

There are many useful approaches that are being applied to well-designed tests and can contribute to the reuse of test design:

Equivalence-Class Partitioning

Boundary-Value Conditions

Strong Data Typing

Design Patterns

Model-Based Testing

Test Oracles

Test-Data Generators

One of the benefits of reusable test design is that you can take advantage of the convergence of these concepts and apply the results, so that they improve the cost/benefit ratio of test efforts on future projects. All of these techniques have elements to contribute to raising test-design abstraction—improving reuse, reducing reinvention, elevating the sophistication of tests, and extending the reach of Software Test Engineers.

Test-Design Reuse Summary

- Reduce ad-hoc reinvention of test designs by capturing and reusing common test patterns.
- Recognize and abstract test issues, and capture them in a form that can be reapplied as a higher level chunk of activity, instead of always dropping down to the detailed instance level.
- Treat past test designs essentially as individual instantiations of a generic test-design "class."
- Institutionalize, "package," or "productize" the results and experience of past test-design work into a reusable deliverable that can be applied to similar test efforts.
- Create checklists of generalized "things to test" that will be a resource for reuse on future versions or applications. The generalized tests augment the separate list of feature details (most often, business rules) and associated tests that truly are unique to a specific project.
- Enumerate common and generic failure modes as items to verify, and make them part of test-design templates.
- Enumerate and document explicit test "coverage" categories, so that they can be incorporated into analysis and test design.
- Ensure that each category contains as complete a set of tests as possible to ensure coverage of the category. Divide the results into generic and unique subsets of tests. Reuse the generic portions of these lists for future efforts.
- Test-design specification templates should be expanded beyond empty-document sections to fill in, to include generic validation rules, test-case checklists, common failure modes, and libraries of generic tests.
- Avoid the pesticide paradox by using a test infrastructure that provides for flexible data-oriented testing at the equivalence-class or data-type level, instead of supporting only specific, handcrafted test cases. Varying the test data and paths of execution will increase the probability of detecting new bugs.

Basic Examples

The following are a few frequently occurring test-design patterns that are suitable for reuse.

CRUD Pattern

1. Identify a record or field upon which to operate (based on input name and parameter info).
2. Generate randomized item from equivalence classes.
3. Verify nonexistence.
4. Add item.
5. Read and verify existence of identical unchanged data.
6. Modify and verify matching modified data.
7. Delete and verify removal of item.

Data-Type Validation Pattern

1. Identify item with type characteristics (for example, a data field) at an abstract level; this should not be limited to simple data types, but should include common business data types (for example, telephone number, address, ZIP code, customer Social Security number, calendar date, and so on).
2. Enumerate the generic business rules that are associated with the type.
3. Define equivalence partitions and boundaries for the values for each business rule.
4. Select test-case values from each equivalence class.

Input Boundary Partition Pattern

1. Enumerate and select an input item.
2. Select a "valid" equivalence partition.
3. Apply a lookup or random generation of a value within that partition, and use it for test.

Stress-Axis Pattern

1. Identify a system response to verify.
2. Identify a stress axis (for example, number of items, size of items, frequency of request, concurrency of requests, complexity of data structures, and dimensionality of input).
3. Identify a starting level of stress.
4. Verify and measure system response.
5. Increase stress, and repeat cycle until system response fails.
6. Switch to another stress axis.
7. Increase the level of that axis until failure.
8. Additionally, add concurrent stress axes.
9. Increase the number of concurrent axes until failure.

Heuristic of Error-Mode Detection Pattern

1. Enumerate past human-error modes.
2. Select a mode that has observed recurrence.
3. Identify a scope in which the failure mode might apply, and routinely test for that failure until you are convinced that it manifested.

File Test Pattern

1. Identify and define files and file semantics to be evaluated.
2. Enumerate failure modes for files.
3. Identify system response for each failure mode to verify (create an Oracle, list).
4. Select each failure mode, and apply it in turn to the designated file.
5. Verify response.

Generic Window UI Operations

1. Open/Close/Maximize/Minimize.
2. Active/Inactive.
3. Validate modality.
4. Validate proper display and behavior when on top or behind.
5. Expand/Resize.
6. Move.

Conclusion

Testers should build test designs around reusable test patterns that are common to a large number of application-testing problems. The inclusion of these patterns as a standard part of the test-design template reduces the time that is spent on test design, by eliminating the need to start with a blank test-design specification. Reuse of these patterns has the added benefit of codifying expert-tester knowledge, so as to increase the likelihood of catching common failures.

source: Mark Folkerts, Tim Lamey, and John Evans - <http://msdn.microsoft.com/en-us/testing/cc514239>.

Application Test Design Patterns

Application Scope Testing Patterns

Extended Use Case Test Pattern

Intent: Develop an application system test suite by modeling essential capabilities as extended use cases.

Context: Extended Use Case test applies when most, if not all, of the essential requirements for the System Under Test can be modeled. Modeling is not always the model of choice for essential system scope capabilities. For example, consider a complex simulation that renders the results with animated, high resolution graphics. The setup and inquiry for a simulation run could be modeled with use cases. However, these use cases could only hint at the test cases. It is possible to develop test cases by imagining specific instances for a use case and corresponding expected results.... More information is available in the source.

source:

Bob Binder - Testing Object-oriented Systems: Models, Patterns, and Tools

<http://www.rbsc.com/pages/TestDesignPatterns.html>

Model Driven Test Pattern

Name: MDT

Intent: Test Objectives

Context: Testing through an API, or with a test automation framework.

Fault Model: Complex sequencing faults

Strategy:

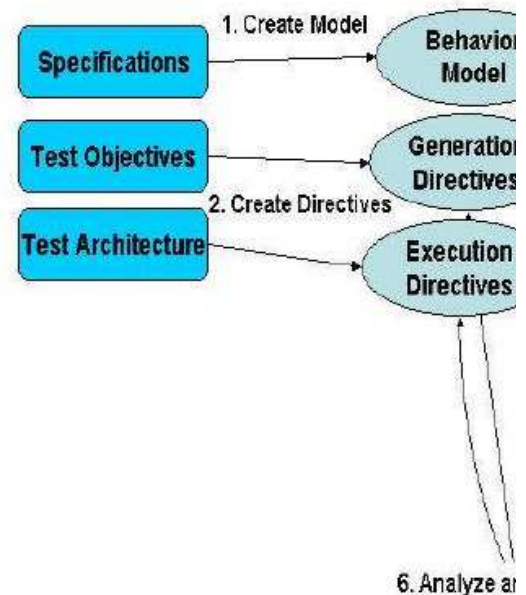
- Model: Step 1
- Procedure: Step 3
- Oracle: From Step 1
- Automate: From Test Architecture

Entry: Model and Architecture well defined

Exit: Analyze and Feedback

Consequences: Upfront investment, thorough testing,

Uses: Huge bibliography



source: Alan Hartman - http://www.imbus.de/qs-day/2007/vortraege/hartman_TestPatterns.pdf

Class Test Design Patterns

Class Scope Testing Patterns

Class scope testing presents test patterns for result-oriented testing at class scope. Class test design must consider method responsibilities, intraclass interactions, and superclass/subclass interactions.

Category-Partition Test Pattern

Intent: Design test suites for methods based on input-output relationships

Context: A systematic technique to identify test cases for a single method based on input-output pairs. Appropriate for cohesive functions that depend on multiple inputs to make decisions or computations.

More information: [TestPatternCategoryPartition.pdf](#)

source:

Bob Binder - Testing Object-oriented Systems: Models, Patterns, and Tools

<http://www.rbsc.com/pages/TestDesignPatterns.html>

Alan Hartman - http://www.imbus.de/qs-day/2007/vortraege/hartman_TestPatterns.pdf

Percolation Test Pattern

Intent: Implement automatic checking of superclass assertions to support design-by-contract and the Liskov substitution principle.

Context:The percolation pattern allows base class invariants, preconditions, and postconditions to be checked in derived class functions without the use of redundant code. It respects base class encapsulation by providing a fixed interface to base class assertions without direct access to base class instance variables. It allows loosely coupled, compile-time enabling/disabling of assertion checking.

More information: [Percolation.pdf](#)

source:

Bob Binder - Testing Object-oriented Systems: Models, Patterns, and Tools

<http://www.rbsc.com/pages/HarnessPatternList.htm>

Gang of Four Test Design Patterns

Gang of Four Test Patterns

The Observer Test Pattern

A portion of the design of an application has been created using the Observer pattern. Our intent is to design effective and efficient interaction test software that tests the completeness of the notifications sent out by the Subject to the Observer, the correctness of the queries sent by the Observers, and the consistency of the data in the Subject and in the Observers.

An ObserverTest object will "observe" the Subject and the Observer to determine whether each test case passes or fails. The ObserverTest pattern is an extension of the PACT generative pattern language [5]. There is a need to test the interactions between the Subject and Observer objects. Both the Subject and Observer implementations have been tested in isolation; however, the interaction between the two has not been tested and can be fairly complex. The Subject may not send notifications to its Observers when it should. The Observer may not send the correct queries of the Subject.

The tests will be conducted within the context of an application whose design is the composition of the Observer pattern and several other design patterns. Some of the Subject and Observer objects may play roles in multiple design patterns. This implies that the ObserverTest pattern may only engage a portion of the interface of each class. The test code does not require any modification of either the Subject or the Observers and can focus on the behaviors and state of the objects that are specific to the Observer pattern. There are several forces that constrain the design of the test software and the specification of the ObserverTest class.

Modifying the production software to test it requires additional tests after the production software is returned to its original form. Therefore, we prefer not to modify the software.

The ObserverTest object must know when the Subject object sends out notifications. The most efficient way to achieve this is to use the test object as an Observer object. Adding the ObserverTest object as an additional Observer to the Subject does not modify the behavior of the production software since it is supposed to be designed to accept an indeterminate number of Observers.

Having a single test object that manages, accesses, and verifies the state of both the Subject and Observer simplifies the need to verify that the state of the Subject object is properly set.

The structure of this pattern is shown in Figure 5. I will Build a Test Class for each Significant Application Class as defined in the PACT pattern language [5]. Define an ObserverTest class for each Subject/Observer pair of classes. Create an ObserverTest object for each ConcreteSubject/ ConcreteObserver association. Register each test object with the ConcreteSubject object as an Observer. Each test object contains a reference to an Observer object. The ObserverTest object invokes the interaction between objects and accesses the state of both objects to determine whether the state of each is correct. The test software will Validate the Test Results Within the Test Case [5] whenever possible.

Include the interface of the Observer interface in the interface of the ObserverTest class. Inherit from the PACT abstract classes. Specify the test suite, test script and test case methods needed to test the interaction between the Subject and Observer. Select test cases based on specific types of test cases.

At the most abstract level there is only one type of test case. In the test case, an ObserverTest object invokes a Subject method that is intended to invoke a state change and a notification. When the ObserverTest object receives notification of the change from the Subject it will check that (1) the appropriate notification is being sent and (2) its associated Observer object has completed the intended action and is now consistent with the Subject. This type of test case produces an actual test case for each state change that is intended to lead to a notification.

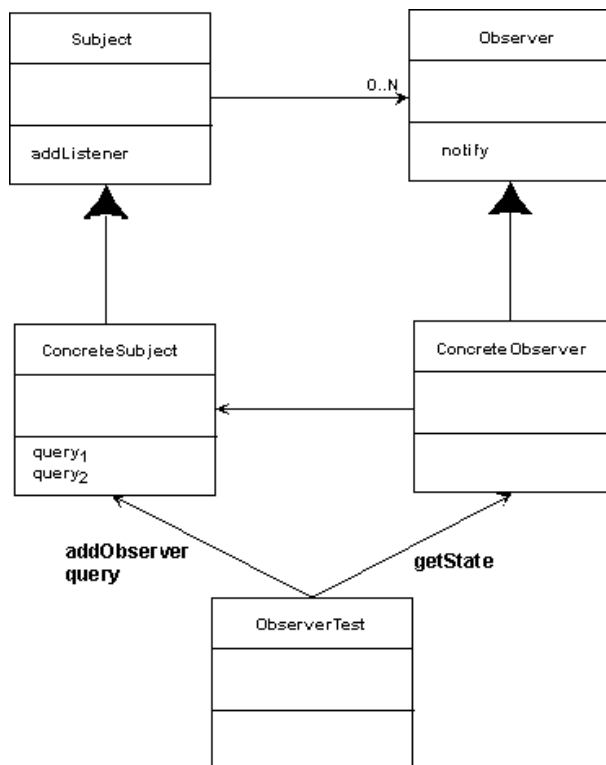


Figure 5 – The ObserverTest Pattern

The ObserverTest object uses the standard interfaces of the Subject for their interaction. Use the addObserver interface to setup the test environment. Specialize the ObserverTest class to each Observer/Subject pair so that it knows which methods in the query interface of the Subject object to use. The specialization is required to represent the unique portion of the state of the Subject that would be requested by the Observer. The ObserverTest object must know the states in which the notification should place the Observer and what messages it was designed to send to the Subject. The ObserverTest object initiates the test sequences in response to one of its test suites being invoked. It creates the Subject and Observer objects, sets up the dependency between specific Subject/Observer objects and then invokes methods on the interface of the Subject in order to induce a notification. The sequence of events is shown in Figure 6.

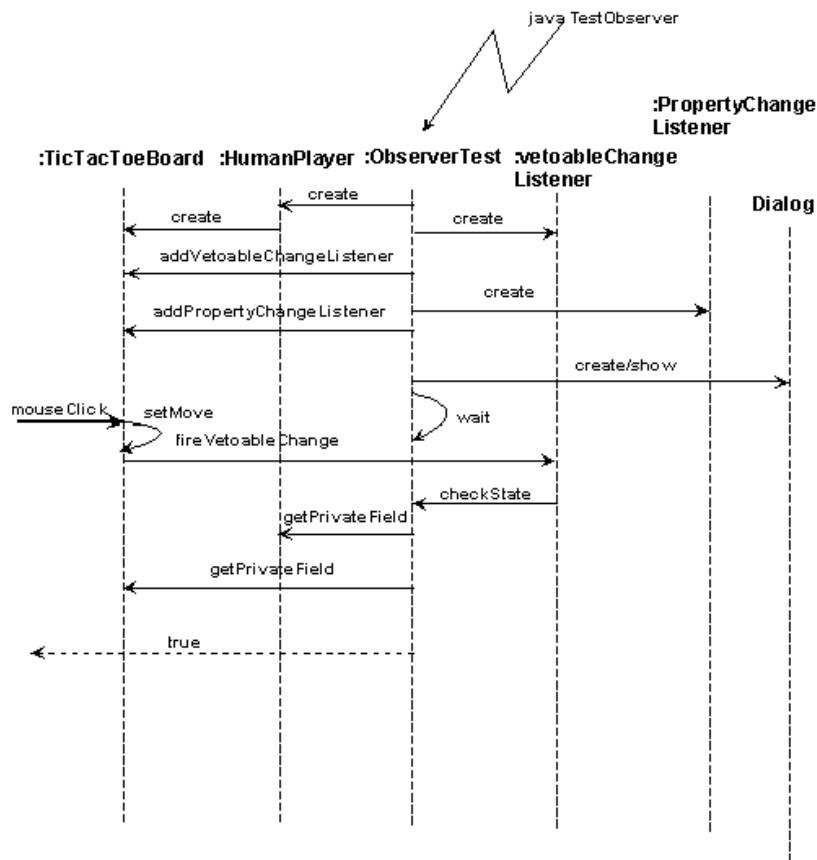


Figure 6 – ObserverTest implementation

The main consequence of applying the ObserverTest pattern is the ability to achieve an "all notifications" level of test coverage. That is, every possible notification is sent in all possible situations. This increases our confidence that the interactions have been properly designed and implemented. Since the Observer pattern is an extensibility device, we also have reason to believe the system is extensible.

The ObserverTest pattern is related to the PACT pattern language. The ObserverTest class is implemented as a PACT class. That implies it encapsulates the test cases needed to test the interactions between the Subject and Observer as well as the logic needed to verify the state of the Subject and Observer objects. As an example, consider applying the canonical test case for the Observer test pattern to code that follows the VetoableChange pattern. The TestObserver adds a VetoableChangeListener for itself with the Subject object. The TestObserver adds a VetoableChangeListener for the Observer object to the Subject. The TestObserver also adds corresponding PropertyChangeListeners. When the TestObserver receives the notification, it checks the state of the Subject and Observer objects. The two objects should be consistent, but in the original state. After the PropertyChange signal is fired, the Subject and Observer objects should be consistent, but in the new state.

Since the VetoableChange pattern is more specific than the Observer pattern, additional test cases can be defined. In particular, a second type of test case can be defined which fires a VetoableChange but results in a PropertyChangeException from the Observer rather than a PropertyChange from the Subject, shown in Figure 6. In this case, the two objects should still be consistent and should still contain the values that were present when the VetoableChange signal was fired. Several actual test cases can be constructed from this type of test case.

Evaluation of the ObserverTest Pattern

Jim Coplien evaluates a pattern on his web page [3] using the criteria that I will use here to evaluate the ObserverTest pattern.

This pattern solves a real problem. Interactions are a source of many errors in component-based software design. Writing effective test software that examines the interactions is difficult. The ObserverTest pattern provides an effective technique for constructing these tests.

The design is a proven concept because it is a variant of the Observer pattern. The test software needs to be notified when an action has happened. The ObserverTest pattern allows the software using the Observer pattern to be tested without being modified.

This solution wasn't obvious (at least to me). I have seen other approaches to this problem used in the field. However, after careful analysis of the forces, this pattern is the optimal solution.

The design describes a relationship between the application software and the test software. The pattern

describes a set of interactions. The test software initiates an interaction, the Subject object treats the ObserverTest object like any other observer, and the ObserverTest object then accesses attributes of the Subject.

The pattern has a significant human component in that it appeals to the inherent search for symmetry. (Incidentally, a variation of the ObserverTest pattern uses multiple observers, one in each process.)

Summary

Analysis of the Product Under Test is the most time-consuming, and therefore most costly, part of testing. Recognizing patterns is an important step in analysis and potentially a major time saver. By reading the design documentation and identifying standard design patterns used by the development team, the tester can apply their own patterns. The development of a test pattern requires the same level of abstraction as is found in the design pattern. The test pattern addresses the fundamental nature of the design pattern while permitting many different implementations. The code, which tests an application of the Observer pattern, will vary greatly from one instance to another. I have shown only one of a number of test patterns that many other testers, and I have used for some time. In future columns I will provide additional patterns and I welcome contributions from others in the testing community.

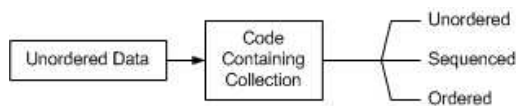
source: John D. McGregor - <http://www.cs.clemson.edu/~johnmc/joop/col18/column18.html>

Unit Test Design Patterns

Collection Management Patterns

A lot of what applications do is manage collections of information. While there are a variety of collections available to the programmer, it is important to verify (and thus document) that the code is using the correct collection. This affects ordering and constraints.

The Collection-Order Pattern



This is a simple pattern that verifies the expected results when given an unordered list. The test validates that the result is as expected:

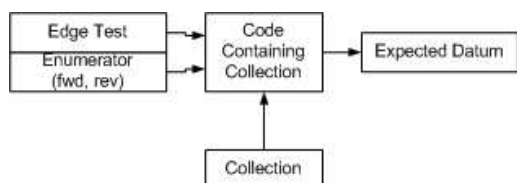
```
unordered
```

```
ordered
```

```
same sequence as input
```

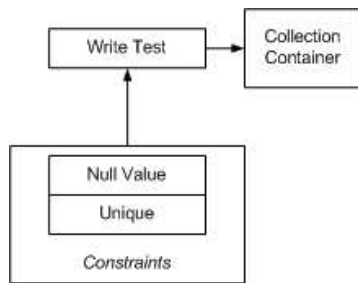
This provides the implementer with crucial information as to how the container is expected to manage the collection.

The Enumeration Pattern



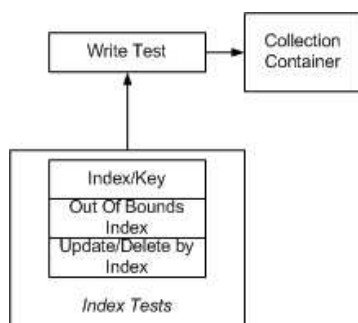
This pattern verifies issues of enumeration, or collection traversal. For example, a collection may need to be traversed forwards and backwards. This is an important test to perform when collections are non-linear, for example a collection of tree nodes. Edge conditions are also important to test--what happens when the collection is enumerated past the first or last item in the collection?

The Collection-Constraint Pattern



This pattern verifies that the container handles constraint violations: null values and inserting duplicate keys. This pattern typically applies only to key-value pair collections.

The Collection-Indexing Pattern



The indexing tests verify and document the indexing methods that the collection container must support--by index and/or by key. In addition, they verify that update and delete transactions that utilize indexing are working properly and are protected against missing indexes.

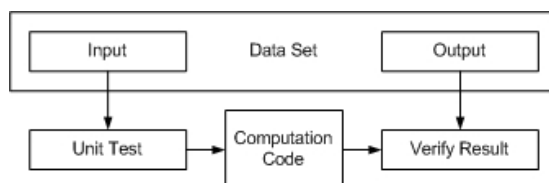
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Data Driven Test Patterns

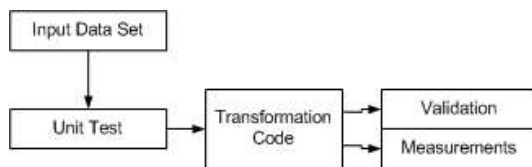
Constructing Parameter-Range unit tests is doable for certain kinds of testing, but it becomes inefficient and complicated to test a piece of code with a complex set of permutations generated by the unit test itself. The data driven test patterns reduce this complexity by separating the test data from the test. The test data can now be generated (which in itself might be a time consuming task) and modified independent of the test.

The Simple-Test-Data Pattern



In the simplest case, a set of test data is iterated through to test the code and a straightforward result (either pass or fail) is expected. Computing the result can be done in the unit test itself or can be supplied with the data set. Variances in the result are not permitted. Examples of this kind of Simple-Test-Data pattern include checksum calculations, mathematical algorithms, and simple business math calculations. More complex examples include encryption algorithms and lossless encoding or compression algorithms.

The Data-Transformation-Test Pattern



The Data-Transformation-Test pattern works with data in which a qualitative measure of the result must be performed. This is typically applied to transformation algorithms such as lossy compression. In this case, for example, the unit test might want to measure the performance of the algorithm with regard to the compression rate vs. the data loss. The unit test may also need to verify that the data can be translated back into something that resembles the input data within some tolerance. There are other applications for this kind of unit test--a rounding algorithm that favors the merchant rather than the customer is a simple example. Another example is precision. Precision occurs frequently in business--the calculation of taxes, interesting, percentages, etc., all of which ultimately must be rounded to the penny or dollar but can have dramatic effects on the resulting value if precision is not managed correctly throughout the calculation.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Data Transaction Patterns

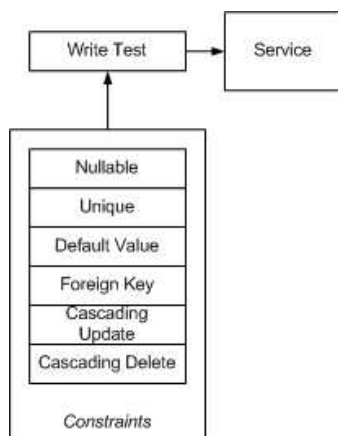
Data transaction patterns are a start at embracing the issues of data persistence and communication. More on this topic is discussed under "Simulation Patterns". Also, these patterns intentionally omit stress testing, for example, loading on the server. This will be discussed under "Stress-Test Patterns".

The Simple-Data-I/O Pattern



This is a simple data transaction pattern, doing little more than verifying the read/write functions of the service. It may be coupled with the Simple-Test-Data pattern so that a set of data can be handed to the service and read back, making the transaction tests a little bit more robust.

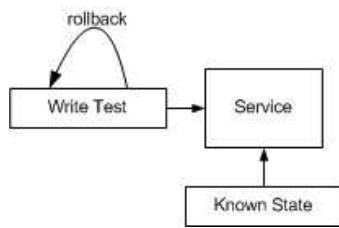
The Constraint-Data Pattern



The Constraint-Data pattern adds robustness to the Simple-Data-I/O pattern by testing more aspects of the service and any rules that the service may incorporate. Constraints typically include: can be null must be unique default value foreign key relationship cascade on update

cascade on delete As the diagram illustrates, these constraints are modeled after those typically found in a database service and are "write" oriented. This unit test is really oriented in verifying the service implementation itself, whether a DB schema, web service, or other model that uses constraints to improve the integrity of the data.

The Rollback Pattern



The rollback pattern is an adjunct to the other transaction testing patterns. While unit tests are supposed to be executed without regard to order, this poses a problem when working with a database or other persistent storage service. One unit test may alter the dataset causing another unit test to inappropriately fail. Most transactional unit tests should incorporate the ability to rollback the dataset to a known state. This may also require setting the dataset into a known state at the beginning of the unit test. For performance reasons, it is probably better to configure the dataset to a known state at the beginning of the test suite rather than in each test and use the service's rollback function to restore that state for each test (assuming the service provides rollback capability).

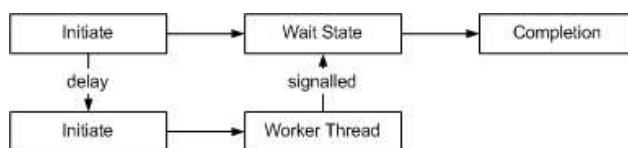
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Multithreading Patterns

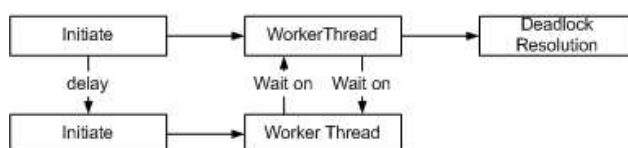
Unit testing multithreaded applications is probably one of the most difficult things to do because you have to set up a condition that by its very nature is intended to be asynchronous and therefore non-deterministic. This topic is probably a major article in itself, so I will provide only a very generic pattern here. Furthermore, to perform many threading tests correctly, the unit tester application must itself execute tests as separate threads so that the unit tester isn't disabled when one thread ends up in a wait state.

The Signalled Pattern



This test verifies that a worker thread eventually signals the main thread or another worker thread, which then completes its task. This may be dependent on other services (another good use of mock objects) and the data on which both threads are operating, thus involving other test patterns as well.

The Deadlock-Resolution Pattern



This test, which is probably very complicated to establish because it requires a very thorough understanding of the worker threads, verifies that deadlocks are resolved.

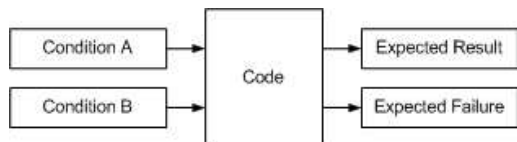
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Pass/Fail Patterns

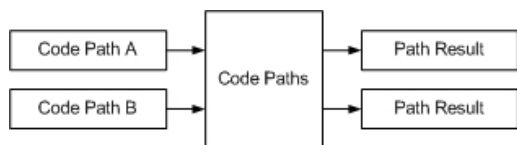
These patterns are your first line of defense (or attack, depending on your perspective) to guarantee good code. But be warned, they are deceptive in what they tell you about the code.

The Simple-Test Pattern



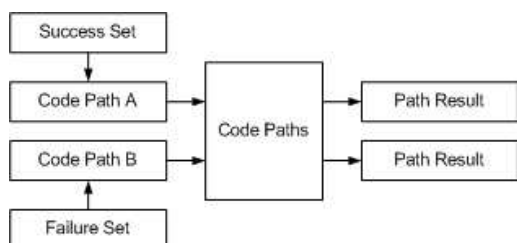
Pass/fail unit tests are the simplest pattern and the pattern that most concerns me regarding the effectiveness of a unit test. When a unit test passes a simple test, all it does is tell me that the code under test will work if I give it exactly the same input as the unit test. A unit test that exercises an error trap is similar--it only tells me that, given the same condition as the unit test, the code will correctly trap the error. In both cases, I have no confidence that the code will work correctly with any other set of conditions, nor that it will correctly trap errors under any other error conditions. This really just basic logic. However, on these grounds you can hear a lot of people shouting "it passed!" as all the nodes on the unit test tree turn green.

The Code-Path Pattern



The Simple-Test pattern typifies what I call "black box testing". Without inspecting the code, that's about all you can do--write educated guesses as to what the code under test might encounter, both as success cases and failure cases, and test for those guesses. A better test ensures that at least all the code paths are exercised. This is part of "white box testing"--knowing the inside workings of the code being tested. Here the priority is not to set up the conditions to test for pass/fail, but rather to set up conditions that test the code paths. The results are then compared to the expected output for the given code path. But now we have a problem--how can you do white box testing (testing the code paths) when the code hasn't been written? Here we are immediately faced with the "design before you code" edge of that sword. The discipline here, and the benefit of unit testing by enforcing some up front design, is that the unit test can test for code paths that the implementer may not typically consider. Furthermore, the unit test documents precisely what the code path is expected to do. Conversely, discipline is needed during implementation when it is discovered that there are code paths that the unit test did not foresee--time to fix the unit test!

The Parameter-Range Pattern



Still, the above test, while improving on the Simple-Test pattern, does nothing to convince me that the code handles a variety of pass/fail conditions. In order to do this, the code should really be

tested using a range of conditions. The Parameter-Range pattern does this by feeding the Code-Path pattern with more than a single parameter set. Now I am finally beginning to have confidence that the code under test can actually work in a variety of environments and conditions.

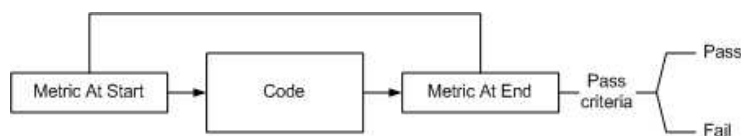
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Performance Patterns

Unit testing should not just be concerned with function but also with form. How efficiently does the code under test perform its function? How fast? How much memory does it use? Does it trade off data insertion for data retrieval effectively? Does it free up resources correctly? These are all things that are under the purview of unit testing. By including performance patterns in the unit test, the implementer has a goal to reach, which results in better code, a better application, and a happier customer.

The Performance-Test Pattern



The basic types of performance that can be measured are:

Memory usage (physical, cache, virtual)

Resource (handle) utilization

Disk utilization (physical, cache)

Algorithm Performance (insertion, retrieval, indexing, and operation)

Note that some languages and operating systems make this information difficult to retrieve. For example, the C# language with its garbage collection is rather difficult to work with in regards to measuring memory utilization. Also, in order to achieve meaningful metrics, this pattern must often be used in conjunction with the Simple-Test-Data pattern so that the metric can measure an entire dataset. Note that just-in-time compilation makes performance measurements difficult, as do environments that are naturally unstable, most notably networks. I discuss the issue of performance and memory instrumentation in my fourth article in a series on advanced unit testing found at <http://www.codeproject.com/csharp/autp4.asp>.

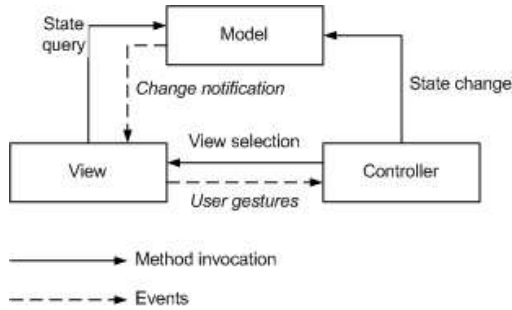
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Presentation Layer Patterns

One of the most challenging aspects of unit testing is verifying that information is getting to the user right at the presentation layer itself and that the internal workings of the application are correctly setting presentation layer state. Often, presentation layers are entangled with business objects, data objects, and control logic. If you're planning on unit testing the presentation layer, you have to realize that a clean separation of concerns is mandatory. Part of the solution involves developing an appropriate Model-View-Controller (MVC) architecture. The MVC architecture provides a means to develop good design practices when working with the presentation layer. However, it is easily abused. A certain amount of discipline is required to ensure that you are, in fact, implementing the MVC architecture correctly, rather than just in word alone. Sun Microsystems has a webpage which I consider is the gospel for the MVC architecture:

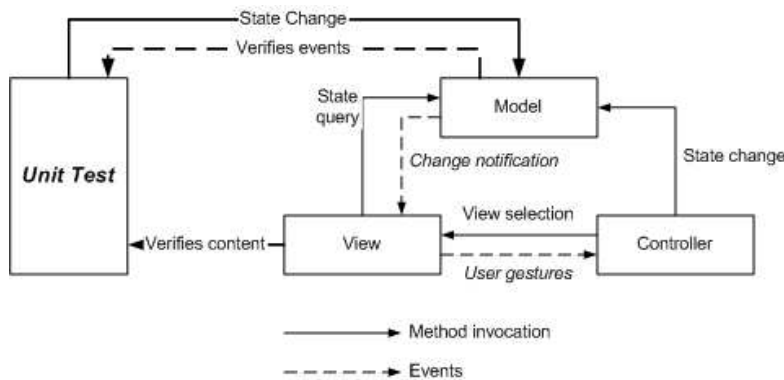
<http://java.sun.com/blueprints/patterns/MVC-detailed.html>. To summarize the MVC pattern here:



It is vital that events are used for model notification changes and user gestures, such as clicking on a button. If you don't use events, the model breaks because you can't easily exchange the view to adjust the presentation to the particular presentation layer requirement. Furthermore, without events, objects become entangled. Also, events, such as managed by an event pool, allow for instrumentation and ease debugging. The only exception to this model that I have found in practice is that on occasion, a state change in the model might be captured by an event in the controller rather than the view. Some model changes, such as user authorization, are view-less but end up affecting other aspects of the controller.

The View-State Test Pattern

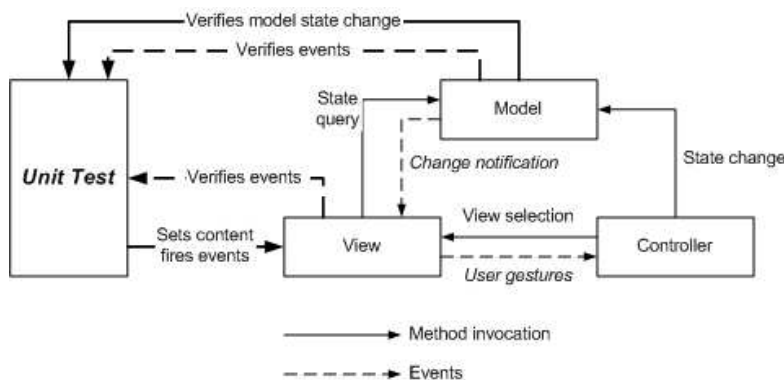
This test verifies that for a change in the model state, the view changes state appropriately.



This test exercises only half of the MVC pattern--the model event notifications to the view, and the view management of those events in terms of affects on the presentation layer content and state. The controller is not a factor in this test pattern.

The Model-State Test Pattern

Once we have verified application's performance with the View-State Pattern, we can progress to a more complicated one. In this pattern, the unit test simulates user gestures by setting state and content directly in the presentation layer, and if necessary, invoking a state change event such as "KeyUp", "Click", etc.



As diagrammed above, the unit test verifies that the model state has changed appropriately and that the expected events fired. In addition, the view's events can be hooked and verified to fire correctly for the simulated user gesture. This test may require some setup on the model itself, and treats the controller as a black box, however model state can be inspected to determine whether the controller is managing the model state appropriately.

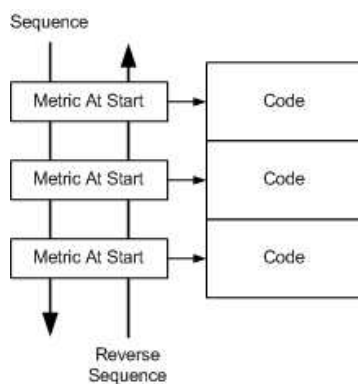
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Process Patterns

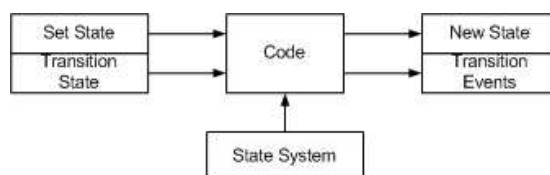
Unit testing is intended to test, well, units...the basic functions of the application. It can be argued that testing processes should be relegated to the acceptance test procedures, however I don't buy into this argument. A process is just a different type of unit. Testing processes with a unit tester provide the same advantages as other unit testing--it documents the way the process is intended to work and the unit tester can aid the implementer by also testing the process out of sequence, rapidly identifying potential user interface issues as well. The term "process" also includes state transitions and business rules, both of which must be validated.

The Process-Sequence Pattern



This pattern verifies the expected behavior when the code is performed in sequence, and it validates that problems when code is executed out of sequence are properly trapped. The Process-Sequence pattern also applies to the Data-Transaction pattern--rather than forcing a rollback, resetting the dataset, or loading in a completely new dataset, a process can build on the work of the previous step, improving performance and maintainability of the unit test structure.

The Process-State Pattern



The concept of state cannot be decoupled from that of process. The whole point of managing state is so that the process can transition smoothly from one state to another, performing any desired activity. Especially in "stateless" systems such as web applications, the concept of state (as in the state of the session) is important to test. To accomplish this without a complicated client-server setup and manual actions requires a unit tester that can understand states and allowable transitions and possibly also work with mock objects to simulate complicated client-server environments.

The Process-Rule Pattern



This test is similar to the Code-Path pattern--the intention is to verify each business rule in the system. To implement such a test, business rules really need to be properly decoupled from surrounding code--they cannot be embedded in the presentation or data access layers. As I state elsewhere, this is simply good coding, but I'm constantly amazed at how much code I come across that violates these simple guidelines, resulting in code that is very difficult to test in discrete units. Note that here is another benefit of unit testing--it enforces a high level of modularity and decoupling.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

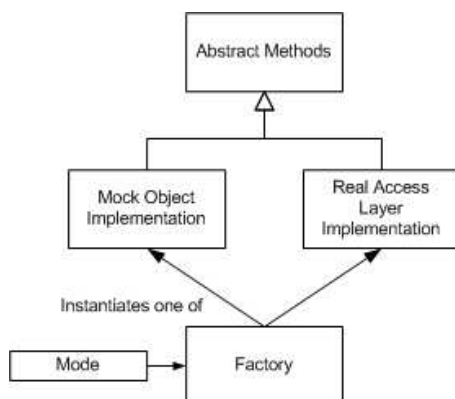
Unit Test Design Patterns

Simulation Patterns

Data transactions are difficult to test because they often require a preset configuration, an open connection, and/or an online device (to name a few). Mock objects can come to the rescue by simulating the database, web service, user event, connection, and/or hardware with which the code is transacting. Mock objects also have the ability to create failure conditions that are very difficult to reproduce in the real world--a lossy connection, a slow server, a failed network hub, etc. However, to properly use mock objects the code must make use of certain factory patterns to instantiate the correct instance--either the real thing or the simulation. All too often I have seen code that creates a database connection and fires off an SQL statement to a database, all embedded in the presentation or business layer! This kind of code makes it impossible to simulate without all the supporting systems--a preconfigured database, a database server, a connection to the database, etc. Furthermore, testing the result of the data transaction requires another transaction, creating another failure point. As much as possible, a unit test should not in itself be subject to failures outside of the code it is trying to test.

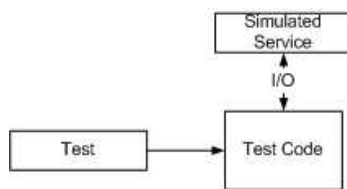
Mock-Object Pattern

In order to properly use mock objects, a factory pattern must be used to instantiate the service connection, and a base class must be used so that all interactions with the service can be managed using virtual methods. (Yes, alternatively, Aspect Oriented Programming practices can be used to establish a pointcut, but AOP is not available in many languages). The basic model is this:



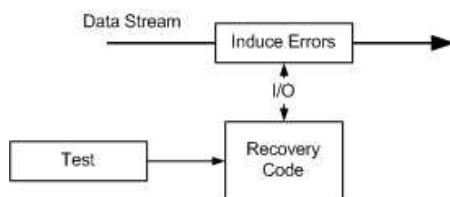
To achieve this construct, a certain amount of foresight and discipline is needed in the coding process. Classes need to be abstracted, objects must be constructed in factories rather than directly instantiated in code, facades and bridges need to be used to support abstraction, and data transactions need to be extracted from the presentation and business layers. These are good programming practices to begin with and result in a more flexible and modular implementation. The flexibility to simulate and test complicated transactions and failure conditions gains a further advantage to the programmer when mock objects are used.

The Service-Simulation Pattern



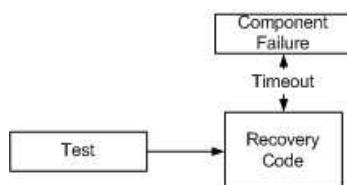
This test simulates the connection and I/O methods of a service. In addition to simulating an existing service, this pattern is useful when developing large applications in which functional pieces are yet to be implemented.

The Bit-Error-Simulation Pattern



I have only used this pattern in limited applications such as simulating bit errors induced by rain-fade in satellite communications. However, it is important to at least consider where errors are going to be handled in the data stream--are they handled by the transport layer or by higher level code? If you're writing a transport layer, then this is a very relevant test pattern.

The Component-Simulation Pattern



In this pattern, the mock object simulates a component failure, such as a network cable, hub, or other device. After a suitable time, the mock object can do a variety of things:

```

throw an exception
return incomplete or completely missing data
return a "timeout" error
  
```

Again, this unit test documents that the code under test needs to handle these conditions.

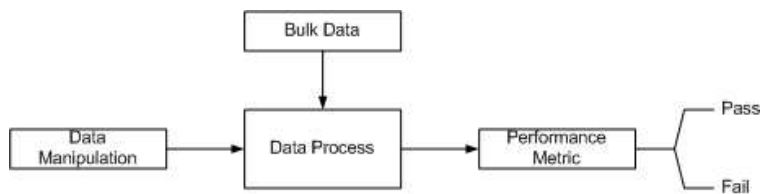
source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Stress-Test Patterns

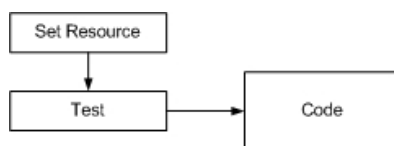
Most applications are tested in ideal environments--the programmer is using a fast machine with little network traffic, using small datasets. The real world is very different. Before something completely breaks, the application may suffer degradation and respond poorly or with errors to the user. Unit tests that verify the code's performance under stress should be met with equal fervor (if not more) than unit tests in an ideal environment.

The Bulk-Data-Stress-Test Pattern



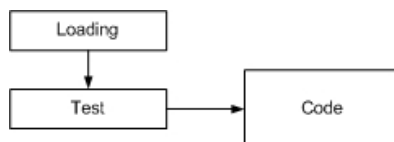
This test is designed to validate the performance of data manipulation when working with large data sets. These tests will often reveal inefficiencies in insertion, access, and deletion processes which are typically corrected by reviewing the indexing, constraints, and structure of the data model, including whether code is should be run on the client or the server.

The Resource-Stress-Test Pattern



Resource consumption stress testing depends on features of the operating system and may be served better by using mock objects. If the operating system supports simulating low memory, low disk space, and other resources, then a simple test can be performed. Otherwise, mock objects must be used to simulate the response of the operating system under a low resource condition.

The Loading-Test Pattern



This test measures the behavior of the code when another machine, application, or thread is loading the "system", for example high CPU usage or network traffic. This is a simulation only (which does not use mock objects) and therefore is of dubious value. Ideally, a unit test that is intended to simulate a high volume of network traffic would create a thread to do just that--inject packets onto the network.

source: Marc Clifton - <http://www.codeproject.com/KB/architecture/autp5.aspx>

Unit Test Design Patterns

Test Harness Design Patterns

The Microsoft® .NET Framework provides you with many ways to write software test automation. But in conversations with my colleagues I discovered that most engineers tend to use only one or two of the many fundamental test harness design patterns available to them. Most often this is true because many developers and testers simply aren't aware that there are more possibilities.

Furthermore I discovered that there is some confusion and debate about when to use a lightweight test harness and when to use a more sophisticated test framework like the popular NUnit. In this month's column James Newkirk, the original author of NUnit, joins me to explain and demonstrate how to use fundamental lightweight test harness patterns and also show you their relation to the more powerful NUnit test framework. The best way to show you where the two of us are headed is with three screen shots. Suppose you are developing a .NET-based application for Windows®. The screen shot in Figure 1 shows the fairly simplistic but representative example of a poker game. The poker application references a PokerLib.dll library that has classes to create and manipulate various poker objects. In particular there is a Hand constructor that accepts a string argument like "Ah Kh Qh Jh Th" (ace of hearts through 10 of hearts) and a Hand.GetHandType method that returns an enumerated type with a string representation like RoyalFlush.



Figure 1 System Under Test

Now suppose you want to test the underlying PokerLib.dll methods for functional correctness. Manually testing the library would be time-consuming, inefficient, error-prone, and tedious. You have two better testing strategies. A first alternative to manual testing is to write a lightweight test harness that reads test case input and expected values from external storage, calls methods in the library, and compares the actual result with the expected result. When using this approach, you can employ one of several basic design patterns. Figure 2 shows a screen shot of a test run that uses the simplest of the design patterns. Notice that there are five test cases included in this run; four cases passed and one failed. The second alternative to manual testing is to use a test framework. Figure 3 shows a screen shot of a test run which uses the NUnit framework.



Figure 2 Lightweight Test Harness Run

In the sections that follow, we will explain fundamental lightweight test harness design patterns, show you a basic NUnit test framework approach, give you guidance on when each technique is most appropriate, and describe how you can adapt each technique to meet your own needs. You'll learn the pros and cons of multiple test design patterns, and this information will be a valuable addition to your developer, tester, and manager skill sets.

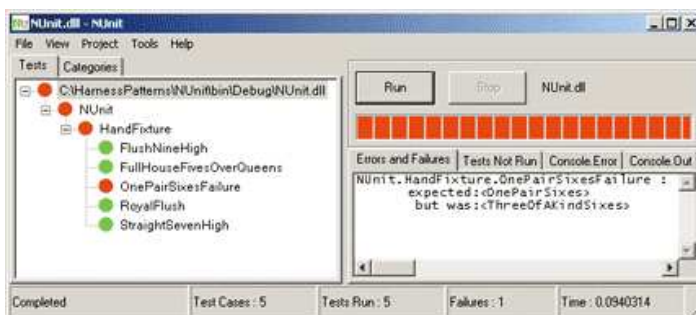


Figure 3 NUnit Test Framework Run

The Six Basic Lightweight Test Harness Patterns

It is useful to classify lightweight data-driven test harness design patterns into six categories based on type of test case storage and test case processing model. There are three fundamental types of test case storage: flat file, hierarchical, and relational. Additionally, there are two fundamental processing models: streaming and buffered. This categorization leads to six test harness design patterns, the cross-product of the storage types with the processing models.

Of course you can think of many other possibilities, but these six categories give you a practical way to think about structuring your lightweight test harnesses. Notice that this assumes that the test case storage is external to the test harness code. In general, external test case storage is better than embedding test case data with the harness code because external storage can be edited and shared more easily than embedded data. However, as we'll explain later, the test-driven approach is primarily a developer activity and typically uses embedded test case data which does have certain advantages over

external data. Separately, NUnit can be used with external test case storage and can support both streaming and buffered processing models.

Flat Test Case Data

The most rudimentary type of test case data is flat data. The data in Figure 4 is the test case file used to generate the test run shown in Figure 2. Compared with hierarchical data and relational data, flat data is most appropriate when you have simple test case input and expected values, you are not in an XML-dominated environment, and you do not have a large test management structure.



Figure 4 Flat Test Case Data

At a minimum every test case has an ID, one or more inputs, and one or more expected results. There is nothing profound about how to store test case data. Examples of flat data are text files, Excel worksheets, and individual tables in a database. Examples of hierarchical data stores are XML files and some .ini files. SQL Server™ databases and Access databases are examples of relational data stores when multiple tables are used in conjunction through relationships. Here you can see we're using a simple text file with a test case ID field, a single input field, and a single expected result field—simple and effective. We will discuss the pros and cons of each of the three storage types later in this column.

This pseudocode shows the basic streaming processing model:

```

open test case data store
loop
  read a test case
  parse id, input, expected
  send input to system under test
  if actual result == expected result
    write pass result to external results file
  else
    write fail result to external results file
  end if
end loop
  
```

The code in Figure 5 shows the main loop. The algorithm is implemented in Visual Basic® .NET, but any .NET-targeted language could be used. The complete source code for all examples is available in the code download that accompanies this column.

```

while ((line = sr.ReadLine()) != null) // main loop
{
  tokens = line.Split(':'); // parse input
  caseid = tokens[0];
  cards = tokens[1].Split(' ');
  expected = tokens[2];

  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]); // test
  actual = h.GetHandType().ToString();
  Console.WriteLine(caseid + " ");
  sw.WriteLine(caseid + " ");
  if (actual == expected) // determine result
  {
    string rv = string.Format(" Pass   {0} = {1}",
      h.ToShortString(), actual);
    Console.WriteLine(rv);
    sw.WriteLine(rv);
  }
  else
  {
    string rv = string.Format(" *FAIL*  actual = {0} expected = {1}",
      actual, expected);
    Console.WriteLine(rv);
    sw.WriteLine(rv);
  }
} //main loop
  
```

Figure 5 Streaming Flat Data Design

Notice that we echo test results to the command shell with a `Console.WriteLine` statement and write test results to an external text file with a call to `StreamWriter.WriteLine`. In general, it makes sense to save test case results to the same type of storage as your test case data, but this is considered to be more a matter of consistency than a technical issue. We call the algorithm a streaming model because it resembles the .NET input-output streaming model; there is a continuous stream of test case input and test results. Now let's look at the buffered model. The pseudocode in Figure 6 is what we'll call

the buffered processing model.

```

open test case data store
loop
  read a test case from external storage
  save test data to in-memory data store
end loop

loop
  read a test case
  parse test case id, input, expected
  send input to system under test
  if actual result == expected result
    write pass result to in-memory data store
  else
    write fail result to in-memory data store
  end if
end loop

loop
  read test result from in-memory store
  write test result to external storage
end loop

```

Figure 6 Buffered Algorithm

With the buffered test harness model we read all test case data into memory before executing any test cases. All test results are saved to an in-memory data store and then emitted to external storage after all test cases have been executed. In other words, test case input and results are buffered through the test system rather than streamed through the system. The code snippet in Figure 7 shows you how we implemented the buffered model using the test case data file that is shown in Figure 4.

```

// 1. read test case data into memory
ArrayList cases = new ArrayList();
string line;
while ((line = sr.ReadLine()) != null) cases.Add(line);

// 2. main test processing loop
ArrayList results = new ArrayList();
string caseid, expected, actual, result;
string[] tokens, cards;
for (int i = 0; i < cases.Count; ++i)
{
  tokens = cases[i].ToString().Split(':'); // parse input
  caseid = tokens[0];
  cards = tokens[1].Split(' ');
  expected = tokens[2];

  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();
  result = caseid + " " + (actual == expected ?
    " Pass " + h.ToShortString() + " = " + actual :
    " *FAIL* actual = " + actual + " expected = " + expected);
  results.Add(result); // store result into memory
}

// 3. emit results to external storage
for (int i = 0; i < results.Count; ++i)
{
  Console.WriteLine(results[i].ToString());
  sw.WriteLine(results[i]);
}

```

Figure 7 Flat Data Buffered Design

If you compare the streaming processing model with the buffered model, it's pretty clear that the streaming model is both simpler and shorter. So why would you ever want to use the buffered model? There are two common testing scenarios where you should consider using the buffered processing model instead of the streaming model. First, if the aspect in the system under test involves file input/output, you often want to minimize the test harness file operations. This is especially true if you are monitoring performance. Second, if you need to perform some pre-processing of your test case input or post-processing of your test case results (for example aggregating various test case category results), it's almost always more convenient to have all results in memory where you can process them. The NUnit test framework is very flexible and can use external test case storage primarily with the buffered processing models. However, a complete discussion of how to use NUnit in these ways would require an entire article by itself and is outside the scope of this column.

Hierarchical Test Case Data

Hierarchical test case data, especially XML, has become very common. In this section we will show you the streaming and buffered lightweight test harness processing models when used in conjunction with XML test case data. Compared with flat test case data and relational data, hierarchical XML-based test case data is most appropriate when you have relatively complex test case input or expected results, or you are in an XML-based environment (your development and test effort infrastructure relies heavily on XML technologies). Here is a sample of XML-based test case data that corresponds to the flat file test case data in Figure 4:

```
<?xml version="1.0" ?>
<TestCases>
  <case caseid="0001">
    <input>Ah Kh Qh Jh Th</input>
    <expected>RoyalFlush</expected>
  </case>
  <case caseid="0002">
    <input>Qh Qs 5h 5c 5d</input>
    <expected>FullHouseFivesOverQueens</expected>
  </case>
  ...
</TestCases>
```

Because XML is so flexible there are many hierarchical structures we could have chosen. For example, the same test cases could have been stored as follows:

```
<?xml version="1.0" ?>
<TestCases>
  <case caseid="0001" input="Ah Kh Qh Jh Th" expected="RoyalFlush" />
  <case caseid="0002" input="Qh Qs 5h 5c 5d"
    expected=" FullHouseFivesOverQueens" />
  ...
</TestCases>
```

Just as with flat test case data, you can use a streaming processing model or a buffered model. In each case the algorithm is the same as shown in the basic streaming processing model algorithm and in the buffered algorithm that was shown in Figure 6. Interestingly though, the XML test case data model implementations are quite different from their flat data counterparts. Figure 8 shows key code from a C#-based streaming model implementation.

```
xtr.WriteStartDocument();
xtr.WriteStartElement("TestResults");

while (!xtr.EOF) // main loop
{
  if (xtr.Name == "TestCases" && !xtr.IsStartElement()) break;

  while (xtr.Name != "case" || !xtr.IsStartElement())
    xtr.Read(); // advance to a <case> element if not there yet

  caseid = xtr.GetAttribute("caseid");
  xtr.Read(); // advance to <input>
  input = xtr.ReadElementString("input"); // advance to <expected>
  expected = xtr.ReadElementString("expected"); // advance to </case>
  xtr.Read(); // advance to next <case> or </TestCases>

  cards = input.Split(' ');
  Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
  actual = h.GetHandType().ToString();

  xtr.WriteStartElement("result");
  xtr.WriteStartAttribute("caseid", null);
  xtr.WriteString(caseid);
  xtr.WriteEndElement();
  xtr.WriteString(actual == expected ?
    " Pass " + h.ToShortString() + " = " + actual :
    " *FAIL* actual = " + actual + " expected = " + expected);
  xtr.WriteEndElement(); // </result>
} // main loop
```

Figure 8 XML Data Streaming Design

With a streaming model, we use an `XmlTextReader` object to read one XML node at a time. But because XML is hierarchical it is a bit tricky to keep track of exactly where we are within the file, especially when the nested becomes more extreme (in this particular example, the data is little more than a flat file, but it could be significantly more complex). We use an `XmlTextWriter` object to save test results in XML form. Now we'll show you a buffered approach for XML test case data. Figure 9 shows key code from a buffered processing model implementation.

```

// 1. read test case data into memory
XmlSerializer xds = new XmlSerializer(typeof(TestCases));
TestCases tc = (TestCases)xds.Deserialize(sr);

// 2. processing loop
string expected, actual;
string[] cards;
TestResults tr = new TestResults(tc.Items.Length);

for (int i = 0; i < tc.Items.Length; ++i) // test loop
{
    SingleResult res = new SingleResult();
    res.caseid = tc.Items[i].caseid;
    cards = tc.Items[i].input.Split(' ');
    expected = tc.Items[i].expected;

    Hand h = new Hand(cards[0], cards[1], cards[2], cards[3], cards[4]);
    actual = h.GetHandType().ToString();

    res.result = (actual == expected) ? // store results into memory
        "Pass " + h.ToShortString() + " = " + actual :
        "*FAIL*" + "actual = " + actual + " expected = " + expected;
    tr.Items[i] = res;
}

// 3. emit results to external storage
XmlTextWriter xtw = new XmlTextWriter("../\\TestResults.xml", System.Text.Encoding.UTF8);
XmlSerializer xs = new XmlSerializer(typeof(TestResults));
xs.Serialize(xtw, tr);

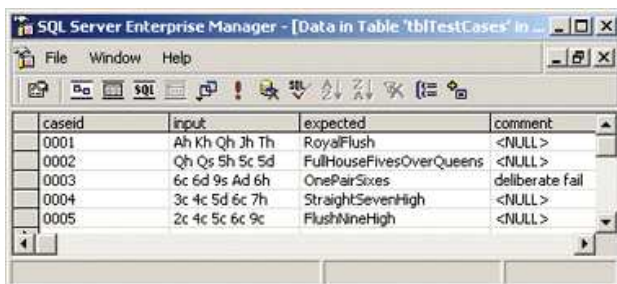
```

Figure 9 XML Data Buffered Design

We use an `XmlSerializer` object from the `System.Xml.Serialization` namespace to read the entire XML test case file into memory with a single line of code and also to write the entire XML result file with a single line of code. Of course, this requires us to prepare appropriate collection classes (`TestCases` and `TestResults` in the code) to hold the data. Unlike flat test case data, with XML data the buffered model test harness code tends to be shorter and simpler. So when might you consider using a streaming model in conjunction with XML test case data? Most often you will want to use a streaming model when you have a lot of test cases to deal with. Reading a huge amount of test case data into memory all at once may not always be possible, especially if you are running stress tests under conditions of reduced internal memory.

Relational Test Case Data

In this section we'll describe the streaming and buffered lightweight test harness processing models when used in conjunction with SQL test case data. Compared with flat data and hierarchical data, relational SQL-based test case data is most appropriate when you have a very large number of test cases, or you are in a relatively long product cycle (because you will end up having to store lots of test results), or you are working in a relatively sophisticated development and test infrastructure (because you will have lots of test management tools). Figure 10 shows test case data that has been stored in a SQL database.



caseid	input	expected	comment
0001	Ah Kh Qh Jh Th	RoyalFlush	<NULL>
0002	Qh Qs 5h 5c 5d	FullHouseFivesOverQueens	<NULL>
0003	6c 6d 9s Ad 6h	OnePairSixes	deliberate fail
0004	3c 4c 5d 6c 7h	StraightSevenHigh	<NULL>
0005	2c 4c 5c 6c 9c	FlushNineHigh	<NULL>

Figure 10 SQL-based Test Case Data

Just as with flat test case data and hierarchical data, you can use a streaming processing model or a buffered model. The basic streaming and buffered models described earlier will be the same. The streaming model implementation is included in this column's download. If you examine the code you'll see that for a streaming model we like to use a `SqlDataReader` object and its `Read` method. For consistency we insert test results into a SQL table rather than save to a text file or XML file. We prefer to use two SQL connections—one to read test case data and one to insert test results. As with all the techniques in this column, there are many alternatives available to you.

The code for the buffered processing model can be downloaded from the MSDN Magazine Web site. Briefly, we connect to the test case database, fill a `DataSet` with all the test case data, iterate through each case, test, store all results into a second `DataSet`, and finally emit all results to a SQL table.

Using relational test case data in conjunction with ADO.NET provides you with many options. Assuming memory limits allow, we typically prefer to read all test case data into a `DataSet` object. Because all the test case data is in a single table, we could also have avoided the relatively expensive overhead of a `DataSet` by just using a `DataTable` object. However in situations where your test case data is contained in multiple tables, reading into a `DataSet` gives you an easy way to manipulate test case data using a `DataRelation` object. Similarly, to hold test case results we create a second `DataSet` object

and a DataTable object. After running all the test cases we open a connection to the database that holds the results table (in this example it's the same database that holds the test case data) and write results using the SqlDataAdapter.Update method. Recall that when using flat test case data, a streaming processing model tends to be simpler than a buffered model, but that when using hierarchical XML data, the opposite is usually true. When using test case data stored in a single table in SQL Server, a streaming processing model tends to be simpler than a buffered model and the technique of choice. When test case data spans multiple tables, you'll likely want to use a buffered processing model.

The TDD Approach with the NUnit Framework

In the previous sections you've seen six closely related lightweight test harness design patterns. A significantly different but complementary approach is to use an existing test framework. The best known framework for use in a .NET environment is the elegant NUnit framework as shown in Figure 3. See the MSDN@Magazine article by James Newkirk and Will Stott at Test-Driven C#: Improve the Design and Flexibility of Your Project with Extreme Programming Techniques for details. The code snippet in Figure 11 shows how you can use NUnit to create a DLL that can be used by NUnit's GUI interface. And the code snippet in Figure 12 shows how you can use NUnit with external XML test case data to create a DLL that can be used by NUnit's command-line interface.

```
[Suite]
public static TestSuite Suite
{
    get
    {
        TestSuite testSuite = new TestSuite("XML Buffered Example");
        using(StreamReader reader = new StreamReader("TestCases.xml"))
        {
            XmlSerializer xds = new XmlSerializer(typeof(TestCases));
            TestCases testCases = (TestCases)xds.Deserialize(reader);

            foreach(Case testCase in testCases.cases)
            {
                string[] cards = testCase.input.Split(' ');
                HandType expectedHandType = (HandType)Enum.Parse(
                    typeof(HandType), testCase.expected);
                Hand hand = new Hand(cards[0], cards[1], cards[2],
                    cards[3], cards[4]);
                testSuite.Add(new HandTypeFixture(
                    testCase.id, expectedHandType, hand.GetHandType()));
            }
        }
        return testSuite;
    }
}
```

Figure 12 NUnit Approach with External XML Test Cases

```
using NUnit.Framework;
using PokerLib;

[TestFixture]
public class HandFixture
{
    [Test]
    public void RoyalFlush()
    {
        Hand hand = new Hand("Ah", "Kh", "Qh", "Jh", "Th");
        Assert.AreEqual(HandType.RoyalFlush, hand.GetHandType());
    }
    ... // other tests here
}
```

Figure 11 NUnit Approach with Embedded Test Cases

You may be wondering whether it's better to use NUnit or to write a custom test harness. The best answer is that it really depends on your scenarios and environment, but using both test techniques together ensures a thorough test effort. The NUnit framework and lightweight test harnesses are designed for different testing situations. NUnit was specifically designed to perform unit testing in a test-driven development (TDD) environment, and it is a very powerful tool. A lightweight test harness is useful in a wide range of situations, such as when integrated into the build process, and is more traditional than the NUnit framework in the sense that a custom harness assumes a conventional spiral-type software development process (code, test, fix).

A consequence of NUnit's TDD philosophy is that test case data is typically embedded with the code under test. Although embedded test case data cannot easily be shared (for example when you want to test across different system configurations), embedded data has the advantage of being tightly coupled with the code it's designed to test, which makes your test management process easier. Test-driven development with NUnit helps you write code and test it. This is why embedded tests with NUnit are acceptable—you change your tests as you change your code. Now this is not to say that the two test approaches are mutually exclusive; in particular NUnit works nicely in a code-first, test-later environment, and can utilize an external test case data source. And a lightweight test harness can be used in conjunction with a TDD philosophy. The NUnit framework and custom lightweight test harnesses have different strengths and weaknesses. Some of NUnit's strengths are that it is a solid, stable tool, it is a nearly a de-facto standard because of widespread use, and it has lots of features. The strengths of custom test harnesses are that they are very flexible, allowing you to use internal or external

storage in a variety of environments, test for functionality as well as performance, stress, security and other types of testing, and execute sets of individual test cases or multiple state change test scenarios.

Conclusion

Let's briefly summarize. When writing a data-driven lightweight test harness in a .NET environment you can choose one of three types of external test case data storage: flat data (typically a text file), hierarchical data (typically an XML file), or relational data (typically a SQL Server database). Often you will have no choice about the type of data store to use because you will be working in an already existing development environment. Flat data is good for simple test case scenarios, hierarchical data works very well for technically complex test case scenarios, and relational data is best for large test efforts. When writing a lightweight test harness you can employ either a streaming processing model or you can choose a buffered processing model. A streaming processing model is usually simpler except when used with truly hierarchical or relational data, in which case the opposite is true. A streaming model is useful when you have a very large number of test cases, and a buffered model is most appropriate when you are testing for performance or when you need to process test cases and results. Using a test framework like NUnit is particularly powerful for unit testing when you are employing a TDD philosophy. With the .NET environment and powerful .NET-based tools like NUnit, it's possible to write great test automation quickly and efficiently. The release of Visual Studio® 2005 will only enhance your ability to write test automation and the Team System version in Visual Studio 2005 will have many NUnit-like features. With software systems increasing in complexity, testing is more important than ever. Knowledge of these test harness patterns as well as of frameworks like NUnit will help you test better and produce better software systems.

source: James McCaffrey and James Newkirk - <http://msdn.microsoft.com/en-us/magazine/cc163752.aspx>