Patterns Of Software Test (PoST) – an Overview

Based on the PoST workshops

Keith Stobie

Microsoft

kstobie@acm.org

(work originally started while sponsored by BEA systems.)

Who knows about software patterns?

Which ones?

What are patterns?

What are patterns?

Patterns are a way of helping designers.

” each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”

- **vocabulary** for problem-solvers
- focus attention on the **forces** that make up a problem.
- encourage **iterative** thinking.

2

“A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts” "*Understanding and Using Patterns in Software Development*",

From the Patterns Definitions section of the Patterns Home Page:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves.

Why patterns?

Patterns are a way of helping designers. Christopher Alexander writes in **The Timeless Way of Building**, that "each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution." When done well, patterns accomplish at least three things:

- They provide a vocabulary for problem-solvers. "Hey, you know, we should use a Null Object."
- They focus attention on the forces that make up a problem. That allows designers to better understand when and why a solution applies.
- They encourage iterative thinking. Each solution creates a new context in which new problems can be solved.

History of Patterns

1964-79 formalized by the architect Christopher Alexander.

1987, Ward Cunningham and Kent Beck presented at OOPSLA'87 the paper *"Using Pattern Languages for Object-Oriented Programs"*.

use some of Alexander's ideas for a small five pattern language for guiding novice Smalltalk programmers.

1991, Jim Coplien compiles a catalog of C++ *idioms* published as **Advanced C++ Programming Styles and Idioms**.

1991 Discussions of patterns abounded at OOPSLA'91 at a workshop.

1993, Kent Beck & Grady Booch sponsor a mountain retreat in Colorado, 1st meeting of the Hillside Group & patterns workshop at OOPSLA'93

1994, **Hillside Group** met again to plan the first PLoP conference.

1995, the *[GoF]* **Design Patterns** book was published

. . .

2001, PoST workshops

Feb02 SASQAG Microsoft 3

In 1987, Ward Cunningham and Kent Beck were working with Smalltalk and designing user interfaces. They decided to use some of Alexander's ideas to develop a small five pattern language for guiding novice Smalltalk programmers. They wrote up the results and presented them at OOPSLA'87 in Orlando in the paper *"Using Pattern Languages for Object-Oriented Programs"*.

Soon afterward, Jim Coplien (more affectionately referred to as "Cope") began compiling a catalog of C++ *idioms* (which are one kind of pattern) and later published them as a book in 1991, **Advanced C++ Programming Styles and Idioms**.

From 1990 to 1992, various members of the Gang of Four had met one another and had done some work compiling a catalog of patterns. Discussions of patterns abounded at OOPSLA'91 at a workshop given by Bruce Andersen (which was repeated in 1992). Many pattern notables participated in these workshops, including Jim Coplien, Doug Lea, Desmond D'Souza, Norm Kerth, Wolfgang Pree, and others.

In August 1993, Kent Beck and Grady Booch sponsored a mountain retreat in Colorado, the first meeting of what is now known as the Hillside Group. Another patterns workshop was held at OOPSLA'93 and then in April of 1994, the Hillside Group met again (this time with Richard Gabriel added to the fold) to plan the first PLoP conference.

Shortly thereafter, the *[GoF]* **Design Patterns** book was published, and the rest, is history.

[from: **Brad Appleton** *http://www.enteract.com/~bradapp/*]

Architecture - Alexander

The term "pattern" is derived from the writings of the architect Christopher Alexander as it relates to urban planning and building architecture.

```
IF   you find yourself in CONTEXT
          for example EXAMPLES,
          with PROBLEM,
          entailing FORCES
     THEN for some REASONS,
          apply DESIGN FORM AND/OR RULE
          to construct SOLUTION
          leading to NEW CONTEXT & OTHER PATTERNS
```

Feb02 SASQAG Microsoft 4

The current use of the term "pattern" is derived from the writings of the architect Christopher Alexander who has written several books on the topic as it relates to urban planning and building architecture:

•**Notes on the Synthesis of Form**, Harvard University Press, 1964
(hereafter referred to as "*[Notes]*")

•**The Oregon Experiment**, Oxford University Press, 1975
(hereafter referred to as "*[Oregon]*")

•**A Pattern Language: Towns, Buildings, Construction**, Oxford University Press, 1977
(hereafter referred to as "*[APL]*")

•**The Timeless Way of Building**, Oxford University Press, 1979
(hereafter referred to as "*[TTWoB]*")

Window Place - Context

This pattern helps complete the arrangement of the windows given by **ENTRANCE ROOM (130), ZEN VIEW (134), LIGHT ON TWO SIDES OF EVERY ROOM** (159), **STREET WINDOWS** *(164).*

According to the pattern, at least one of the windows in each room needs to be shaped in such a way as to increase its usefulness as a space.

Window Place

✻ ✻ ✻

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them.

Window Place - Conflict

. . .

These kinds of windows which create “places” next to them are not simply luxuries; they are *necessary.* A room which does not have a place like this seldom allows you to feel fully comfortable or perfectly at ease. Indeed, a room without a window place may keep you in a state of perpetual unresolved conflict and tension—slight, perhaps, but definite.

Window Place - Forces

This conflict takes the following form. If the room contains no window which is a "place," a person in the room will be torn between **two forces:**

1. **He wants to sit down and be comfortable.**
2. **He is drawn toward the light.**

Obviously, if the comfortable places—those places in the room where you most want to sit—are away from the windows, there is no way of overcoming this conflict. You see, then, that our love for window "places" is not a luxury but an organic intuition, based on the ***natural desire a person has to let the forces he experiences run free***. A room where you feel truly comfortable will always contain some kind of window place.

Window Place - Solutions

A bay window.
A window seat.
A low sill.
A glazed alcove.

In principle, any window with a reasonably pleasant view can be a window place, provided that it is taken seriously as a space, a volume, not merely treated as a hole in the wall. Any room that people use often should have a window place.

Now, of course, it is hard to give an exact definition of a “place.” Essentially a “place” is a partly enclosed, distinctly Identifiable spot within a room. All of the following can function as “places” in this sense: bay windows, window seats, a low window sill where there is an obvious position for a comfortable armchair, and deep alcoves with windows all around them. To make the concept of a window place more precise, here are some examples of each of these types, together with discussion of the critical features which make each one of them work.

A bay window. A shallow bulge at one end of a room, with windows wrapped around it. It works as a window place because of the greater intensity of light, the views through the side windows, and the fact that you can pull chairs or a sofa up into the bay.

A window seat. More modest. A niche, just deep enough for the seat. It works best for one person, sitting parallel to the window, back to the window frame, or for two people facing each other in this position.

A low sill. The most modest of all. The right sill height for a window place, with a comfortable chair, is very low: **12 to 14** inches. The feeling of enclosure comes from the armchair— best of all, one with a high back and sides.

A glazed alcove. The most elaborate kind of window place: almost like a gazebo or a conservatory, windows all around it, a small room, almost part of the garden.

Window Place - Summary

Therefore:

In every room where you spend any length of time during the day, make at least one window into a "window place."

Make it low and self-contained if there is room for that— **ALCOVES (179)**; keep the sill low — **LOW SILL (222);** put in the exact positions of frames, and mullions, and seats after the window place is framed, according to the view outside — **BUILT-IN SEATS (202), NATURAL DOORS AND WINDOWS (221).** And set the window deep into the wall to soften light around the edges — **DEEP REVEALS (223).** Under a sloping roof, use **DORMER WINDOWS (231)** to make this pattern. .

Software Patterns

Patterns can apply to more than just software design. They are being used for processes, etc.

- *A Development Process Generative Pattern Language*, James Coplien
  http://www1.bell-labs.com/user/cope/Patterns/Process/index.html
- *A Risk Management Catalog*, Alistair Cockburn
  http://members.aol.com/acockburn/riskcata/risktoc.htm
- *CHECKS Pattern Language of Information Integrity*, Ward Cunningham
  http://c2.com/ppr/checks.html

Feb02 SASQAG Microsoft 11

Brad Appleton's "Patterns and Software: Essential Concepts and Terminology".
It's rather long, but nicely set up for skimming.

Doug Lea's Patterns-Discussion FAQ:
http://www.enteract.com/~bradapp/docs/patterns-intro.html
http://g.oswego.edu/dl/pd-FAQ/pd-FAQ.html

Jim Coplien has put the text of a now-out-of-print booklet,
"Software Patterns Management Briefing", online. It's PDF:
http://www1.bell-labs.com/user/cope/Patterns/WhitePaper/SoftwarePatterns.pdf.

The Software Patterns Management Briefing, published in 1996, was an early articulation of the principles, values, and practices behind the pattern discipline. Still timeless today, it is organized around a body of dozens of example patterns. It covers a wide range of topics ranging from pattern forms, to pattern languages, to the history of software patterns, and pattern ethics.

It captured the spirit of the early and emerging directions of software patterns.

The briefing was written for an inexpert audience. No prior knowledge, either of patterns or of software, is necessary to appreciate the principles highlighted in this briefing. I find it to be an excellent first-time introduction to patterns for the curious, and a good overview that helps provide context even for practitioners who have been using patterns for some time.

Software Design Patterns

Design Patterns: elements of reusable object-oriented software.

[GoF] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

“descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context”

- Design patterns represent solutions to problems that arise when developing software within a particular context
  - “Patterns == problem/ solution pairs in a context”
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
  - They are particularly useful for articulating how and why to resolve non-functional forces
- Patterns facilitate reuse of successful software architectures and designs

Feb02 SASQAG Microsoft 12

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation. Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages ....

Bullets from Dr. David L. Levine CS 342: Patterns and Frameworks Introduction

Design Patterns Template

Pattern Name (Scope, Purpose) The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As Other well-known names for the pattern, if any.

Motivation A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

An applicable situation bullet

Structure Class diagram

Participants The classes and/or objects participating in the design pattern and their responsibilities.

Participant Name Responsibility for what

Collaborations How the participants collaborate to carry out their responsibilities. [Collaboration]

Consequences How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

A consequence bullet. Description of consequence

Implementation What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

An implementation Bullet. Description of Bullet

Sample Code and Usage Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk. Program Listing

Known Uses Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

Proxy Pattern

Intent

Provide a surrogate or placeholder for another object to control access to it.

Also Known As

Surrogate

Motivation

. . . to defer the full cost of its creation and initialization until we actually need to use it. . . . large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once

. . . But what do we put in the document in place of the image?

Feb02 SASQAG Microsoft 14

Discussion

Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.

A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.

A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.

A smart proxy interposes additional actions when an object is accessed. Typical uses include:

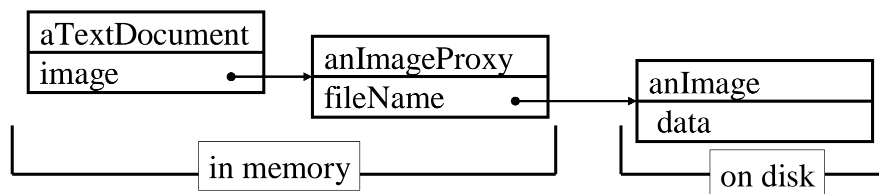
•Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),

•Loading a persistent object into memory when it's first referenced,

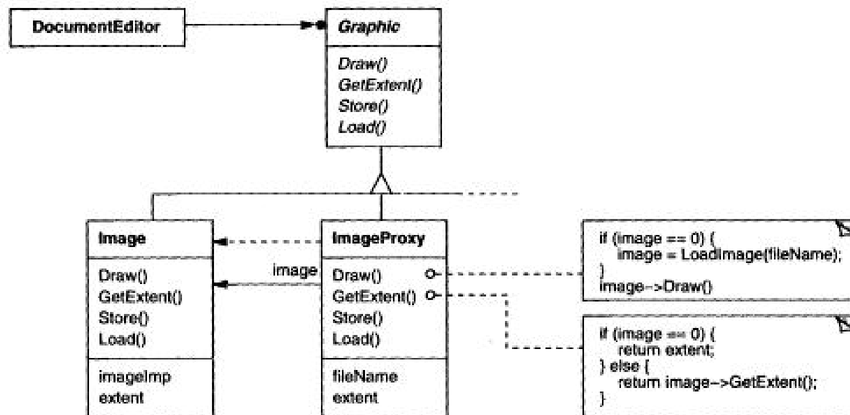
•Checking that the real object is locked before it is accessed to ensure that no other object can change it.

Proxy - Motivation

The solution is to use another object, an image **proxy,** that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



Proxy - Example



Proxy - Applicability

Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy . . .**
   Coplien [Cop92] calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. **. . .**
3. A **protection proxy** controls access to the original object. **. . .**
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed.

Feb02 SASQAG Microsoft 17

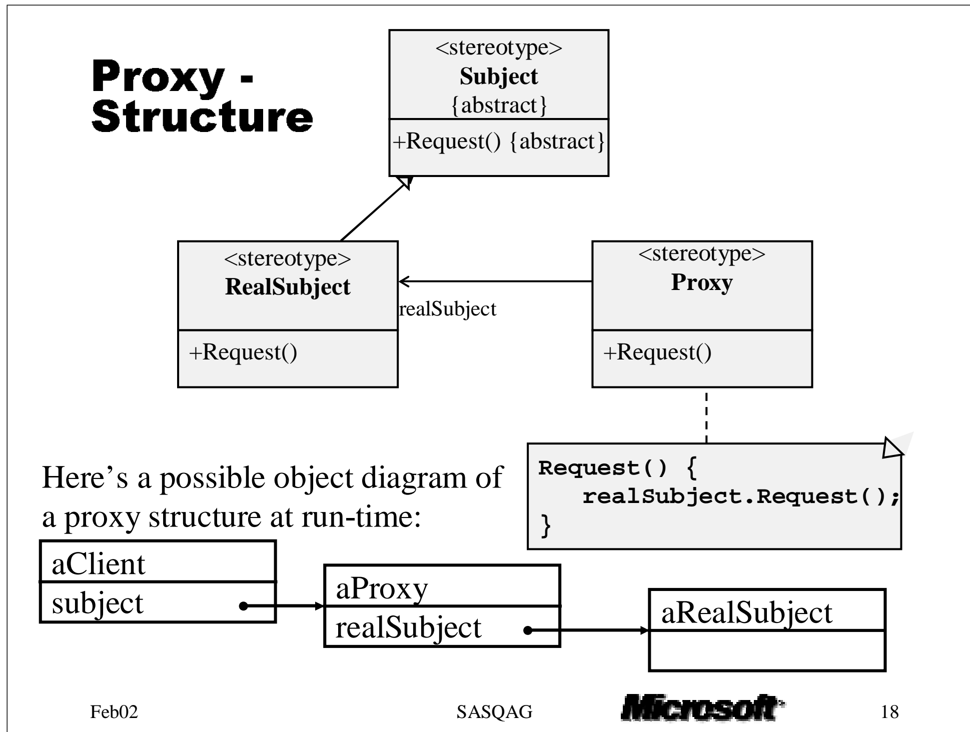
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose. Coplien [Cop92] calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.
4. A **smart reference** is a replacement fora bare pointer that performs additional actions when an object is accessed. Typical uses include

   counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers tEde92I).

   loading a persistent object into memory when it's first referenced.

   checking that the real object is locked before it's accessed to ensure that no other object can change it.



Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account. [Michael Duell, "Non-software examples of software design patterns", *Object Magazine*, Jul 97, p54]

Rules of thumb

Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface. [GOF. p216]

Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests. [GOF, p220]

Proxy - Participants

- Proxy (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. **. . .**
  - provides an interface identical to Subject's **. . .**
  - controls access to the real subject **. . .**
  - other responsibilities depend on the kind of proxy:
    - *remote proxies* , *virtual proxies , protection proxies*
- Subject (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- RealSubject (Image)
  - defines the real object that the proxy represents.

Proxy - Collaborations & Consequences

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. . . .

1. A remote proxy can hide an object in a different address space.
2. A virtual proxy can perform optimizations . . .

Implementation

. . .

Sample Code

Proxy - Uses & Patterns

Known Uses

- Virtual proxy example in Motivation section is from ET++
- NEXTSTEP [Add94] uses proxies . . .
- McCullough [McC87] discusses using proxies in Smalltalk to access remote objects. Pascoe [Pas86] describes how to provide side-effects on method calls & access control with "Encapsulators."

Related Patterns

- Adapter (139): An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. . . .
- Decorator (175): Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.
- Proxies vary in the degree to which they are implemented like a decorator. . . .

Test Patterns?

- *System Test Pattern Language*
  David DeLano and Linda Rising
- *Testing Object-Oriented Systems: Models, Patterns, and Tools*
  Robert Binder
- *Pattern Language for Testing Object-oriented Software*,
  Donald Firesmith, Object Magazine, January 96.
- *Parallel Architecture for Component Testing of Object-Oriented Software*,
  John McGregor & Anuradha Kare, Proceedings of the Ninth International Quality Week, May 96.

Why test patterns?

We believe that testers lack a useful vocabulary, are hampered by rigid "one size fits all" methodologies, and face many problems whose solutions are underdescribed in the literature. Patterns can help with all of those things.

Moreover, the community of pattern writers is a healthy one that regularly spawns new and useful ideas. We testers should link up with it, and we might find its style of work useful as we look for new ideas.

System Test Pattern Language

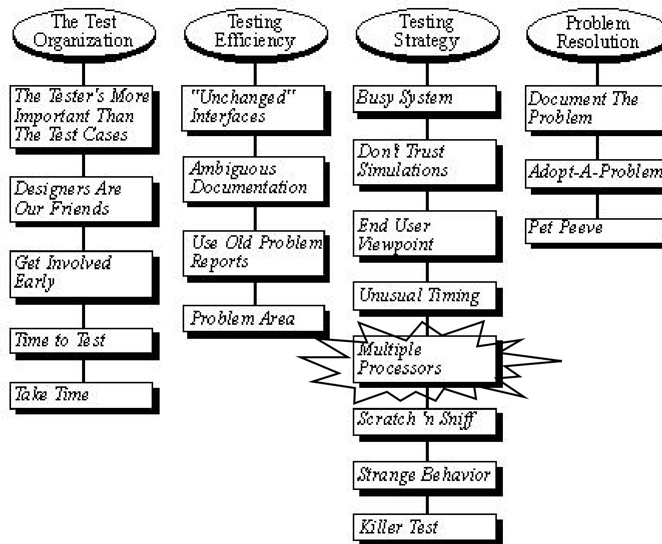
"System Test Pattern Language",
David DeLano and Linda Rising,
AG Communications
http://www.agcs.com/patterns/papers/systestp.htm

- To help System Testers evaluate the readiness of a product for shipping to the customer
- Many of the patterns are orthogonal to all testing phases
- This is not a complete pattern language

Feb02 SASQAG Microsoft 23

“Testing of systems is presently more of an art than a science, even considering current procedures and methodologies that support a more rigorous approach to testing. This becomes even more apparent during the testing of large, embedded systems that have evolved over time. To deliver the best possible product, the role of a System Tester has become vital in the lifecycle of product development. This pattern language has been derived from the experience of veteran System Testers to help System Testers evaluate the readiness of a product for shipping to the customer. Though these patterns have been derived from experience during the system test phase of the product lifecycle, many of the patterns are orthogonal to all testing phases. In addition to System Testers, these patterns may be useful to Designers and Project Managers.

These patterns have been grouped according to their usefulness in the system testing process: The Test Organization, Testing Efficiency, Testing Strategy, and Problem Resolution. This is not a complete pattern language. “



The image below can be used to immediately move to a group of patterns or an individual pattern by clicking on the name of the pattern in which you are interested. It is recommended, however, that the paper be read from top to bottom, instead of moving between the image and each pattern individually.

Pattern: *Multiple Processors*

Problem What strategy should be followed when System Testing a system comprised of multiple processors?

Solution Test across multiple processors.

Resulting Context Problems that occur in one processor will probably occur in other processors. Tests that pass on one processor may fail on another.

Rationale When a problem is found in one processor, that feature will usually have problems running on other processors. A dirty feature is a dirty feature.
Features that run on multiple processors aren't always designed to run on all processors.

Object Oriented Test Patterns

Testing Object-Oriented Systems:
Models, Patterns, and Tools introduces

37 test design patterns

17 design patterns for test automation

16 micro-patterns for test oracles

new pattern template focuses explicitly on key dimensions of test design:

- When is a particular test strategy appropriate?
- What kind of bugs will it find?

Feb02 SASQAG Microsoft 26

Testing Object-Oriented Systems: Models, Patterns, and Tools is a comprehensive guide to testing object-oriented systems. This book introduces testing concepts and shows what is unique about testing object-oriented software. It shows how to develop test models using state machines, combinational logic, and the UML. But it is mainly about test design as software engineering and about the systems engineering challenges of test automation. Over sixty patterns present concepts and techniques for test design and test automation.

Pattern-based Test Design

The book introduces the **test design pattern.** This new pattern template focuses explicitly on the key dimensions of test design: When is a particular test strategy appropriate? What kind of bugs will it find? How do you develop a test suite -- how should the implementation under test be modeled, and how are test cases produced from the model and its oracle? What kind of test automation works best? What are the testing entry and exit criteria? What are its advantages and disadvantages? Who has used this pattern? An overview of the test pattern template explains each of these sections.

The book presents 37 test design patterns based on this template. They cover testing of methods, classes/clusters, subsystems, reusable components, frameworks, and systems. They address responsibility-based testing, integration testing, and regression testing at all these scopes. Seventeen design patterns for test automation support test tool development, and 16 micro-patterns for test oracles show how to produce expected results.

Binder's Test Design Pattern Template

Name
Intent
Context
Fault Model : reached. triggered. propagated.
Strategy : *Test Model, Test Procedure, Oracle, Automation.*
Entry Criteria : test-ready
Exit Criteria
Consequences
Known Uses
Related Patterns

Feb02 SASQAG Microsoft 27

Test Design Pattern Template

Name

A word or phrase that identifies the pattern and suggests its general approach.

Intent

What test design problem does this pattern solve? What is the test strategy? This is a very brief synopsis.

Context

In what circumstances does this pattern apply? To what kind of software entities? At what scope(s)? This section corresponds to the first problem to be solved in test design: given some implementation, what is an effective test design and execution strategy? This section corresponds to the "motivation," "forces," and "applicability" subjects of design patterns.

Category Partition - Context

Intent

Design method scope test suites based on input/output analysis.

Context

How can we develop a test suite to exercise the functions implemented by a single method? **. . .** A systematic technique to identify functions and exercise each input/output relationship is needed.

The Category-Partition pattern is appropriate for any method that implements one or more *independent functions*. **. . .** This approach is qualitative and may be worked by hand. It provides systematic ***implementation-independent*** coverage of method-scope responsibilities.

If the method selects one of many possible responses or has many constraints on parameter values, consider using the ***Combinational Function Test*** pattern.

Context

How can we develop a test suite to exercise the functions implemented by a single method? A method may implement one or several functions, which each may present varying levels of cohesion. Even cohesive functions can have complex input/output relationships. A systematic technique to identify functions and exercise each input/output relationship is needed.

The Category-Partition pattern is appropriate for any method that implements one or more independent functions. For methods or functions that lack cohesion, the constituent responsibilities should be identified and modeled as separate functions. This approach is qualitative and may be worked by hand. It provides systematic implementation-independent coverage of method-scope responsibilities.

If the method selects one of many possible responses or has many constraints on parameter values, consider using the ***Combinational Function Test*** pattern.

Category Partition - Fault Model

Fault Model

The Category-Partition pattern assumes that faults are related to value combinations of message parameters and instance variables and these faults will result in missing or incorrect method output. Offutt and Irvine found that a Category-Partition approach was able to reveal 23 common C++ coding blunders [Offutt+95]. However, faults that are only manifested under certain sequences or which corrupt instance variables hidden by the MUT's interface may not be revealed by category-partition tests.

Fault Model

Why does this pattern work? What kind of bugs does this pattern target? What kinds of bugs are guaranteed to be found if they exist? What kinds of bugs can hide? This section explains how the test model meets the necessary conditions[(2)] for revealing the targeted faults. For a test case to reveal a fault, three things must happen.

- The fault must be **reached**. The test input and state of the system under test must cause the code segment in which the fault is located to be executed.
- The failure must be **triggered**. A fault does not always produce incorrect results when it is reached. The test input and state of the system under test must cause the code segment in which the fault is located to produce incorrect result.
- The failure must be **propagated**. The incorrect result must become observable to the tester or an automated comparator.

This section must explain how the test suite will reach the targeted faults, how it can cause a failure to be triggered, and how a failure will be propagated to become observable.

Category Partition - Test Model

Test Model

Category-partition was first introduced as a general-purpose black-box test design technique for software modules with parameterized functions [Ostrand+88]. Development of a method scope category-partition test suite is illustrated with C++ member function `List::getNextElement( ). This member function ...`

Feb02 SASQAG Microsoft 30

Test Model defines a representation for the responsibilities and/or implementation which are the focus of test design.

Test Model

Category-partition was first introduced as a general-purpose black-box test design technique for software modules with parameterized functions [Ostrand+88]. Development of a method scope category-partition test suite is illustrated with C++ member function `List::getNextElement( )`. This member function returns successive elements of a `List` object. A position in a list of *m* elements is established at the *n*th element by other member functions. A call to `getNextElement( )` returns element $n + 1$. Successive calls to `getNextElement( )` return element $n + 2$, $n + 3$, etc. If no position has been established by a previous operation or the previous position no longer exists due to an intervening change or delete, a `noPosition` exception is thrown. An `emptyList` exception is thrown if the list is empty.

Category Partition - Strategy

Test Procedure

The test suite is produced in seven steps.

1. *Identify the functions of the MUT..*
2. *Identify the input and output parameters of each function.*
3. *Identify* categories *for each input parameter.*
4. *Partition each category into choices.*
   Choices should include legal and illegal values as shown ...
5. *Identify constraints on choices*.
6. *Generate test cases by enumerating all choice combinations.*
7. *Develop expected results for each test case using an appropriate oracle.*
   - Automation

Feb02 SASQAG Microsoft 31

Strategy

This section explains how the test suite is produced and implemented. There are four mandatory subsections.

•*Test Model* defines a representation for the responsibilities and/or implementation which are the focus of test design.

•*Test Procedure* defines an algorithm, technique, or heuristic by which test cases are produced from the model.

•*Oracle* defines the algorithm, technique, or heuristic by which actual results to a test case are to be evaluated for pass/no pass.

•*Automation* discusses automated approaches to test suite generation, test run execution, and test run evaluation. The appropriate extent of manual testing is discussed. In many cases, this section suggests which test automation design patterns are applicable.

The strategy is typically presented by example. The concrete result of applying a test design pattern is an uninteresting, possibly cryptic collection of input/output vectors. Instead of a commentary on test vectors, this section contains a step-by-step application of the test model to suitable example and the resulting test suite.

Category Partition - Criteria

Entry Criteria: Small pop.

Exit Criteria

- Every combination of choices is tested once.
- If the method under test can throw exceptions for incorrect input or other anomalies, the test suite should force each exception at least once.
- Executing the test suite should ***exercise*** at least ***all branches*** in the method under test

Feb02 SASQAG Microsoft 32

Entry Criteria

This section lists preconditions for effective and efficient testing with this pattern. An implementation should not be tested before it is, as a whole, **test-ready**.[(3)] This has three beneficial effects. First, this reduces time lost because the IUT is too buggy to test. Second, this reduces waste of testing resources on bugs which can be more easily revealed and removed by an antecedent process, typically testing at a smaller scope. Third, this reduces the number of bugs that will escape from a given scope of testing. As test scope increases, the extent to which individual components can be exercised decreases. For example, when testing at use case scope, we are typically do not attempt to achieve statement coverage on each component that supports the use case. Instead, we try to achieve coverage of all the inter-component message paths in the use case. We cannot hope to achieve individually adequate coverage when we are testing at higher scope. If lower scope testing is inadequate or skipped, it is likely that many bugs will not be revealed by higher scope testing, even if it is comprehensive. Therefore, meeting the entry criteria will improve both efficiency and effectiveness of testing.[(4)]

Exit Criteria

What objective criteria must be met to achieve a complete test with this pattern? This section defines the extent to which the modeled responsibilities should be exercised and a corresponding coverage metric for an implementation at the scope (e.g., branch coverage at method scope.) or other conditions that indicate adequate testing has been achieved.

Category Partition - Consequences

The Category-Partition pattern is general-purpose, non-quantitative, and straightforward. It does not require advanced analysis or automated support. However, identification of categories and choices is subjective. Skilled and novice testers may identify different categories and choices. Individual blind spots may reduce effectiveness. Bugs that are only manifested under certain sequences of messages to other methods or which corrupt instance variables hidden by the MUT's interface may not be revealed.

The size of a Category-Partition test suite is **...**

The ***Invariant Boundaries*** pattern may be used to produce a smaller test suite whose size is a sum of the number of choices plus one, not the product

With proper attention to interface details, a superclass Category-Partition method test suite may be reused to test an overriding subclass method

Feb02 SASQAG Microsoft 33

Consequences

What are the general prerequisites, costs, benefits, risks, and considerations for using this pattern?

Known Uses

What are the known uses of this test design pattern? What are the known uses of test models and strategies incorporated in this pattern? What are the efficiency and effectiveness of this pattern or similar strategies, as established by empirical studies?

Related Patterns

Are there any test design patterns that are similar or complementary? This section is omitted if there no related patterns.

Category Partition - Uses & Sources

Known Uses

Elements of the Category-Partition approach appear in nearly all black-box test design strategies -- for example, [Myers 79] [Marick 95] [Beizer 95]. A study of manually developed Category-Partition test suites for a small C++ system found 55 out of 75 inserted faults. [Offutt+95].

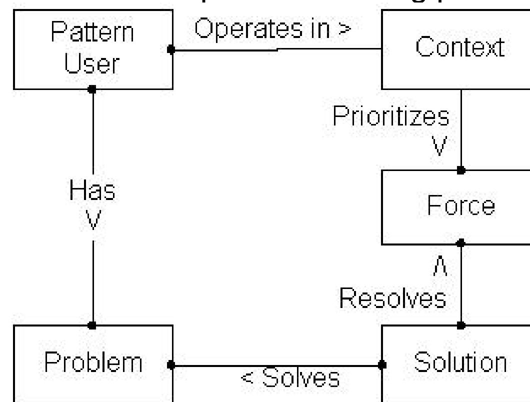
Cited Sources

[Balcer+89] Mark Balcer, William . . .

A Pattern Language for Pattern Writing

by Meszaros and Doble.

http://www.hillside.net/patterns/Writing/patterns.html



Pattern Writing Workshops

PatternLanguagesOfPrograms (PLoP™)

Chili PLoP™

™ PLoP is a trademark of The Hillside Group, Inc

Feb02 SASQAG Microsoft 36

"A Pattern Language for Writer's Workshops"
<http://www1.bell-labs.com/user/cope/Patterns/WritersWorkshops/>

http://www.dreamsongs.com/ChiliPLoPInvite.html

First PoST 's

PoST 1 - Jan. 3-5, 2001 at Rational in Lexington, MA

Introduction by Brian Marick

Pattern reading ("workshopping") demo

Pattern writing & new patterns workshopped.

Attendees: James Bach (co-host), Scott Chase, Sam Guckenheimer (co-host), Elisabeth Hendrickson, Ivar Jacobson, Cem Kaner, Grant Larson, Brian Marick (co-host), Noel Nyman, Bret Pettichord, Johanna Rothman (facilitator), Keith Stobie, Paul Szymkowiak, (co-host), James Tierney, James Whittaker

PoST2 - April 2001 at Florida Institute of Technology, Melbourne, FL

Attendees: Helene Astier, Hans Buwalda, Scott Chase, Elisabeth Hendrickson, Alan A. Jorgenson, Cem Kaner, Brian Marick, Florence Mottay, Melissa Mutkoski, Bret Pettichord, Nadim H. Rabbani, Jennifer Smith-Brock, Keith Stobie, Amit Singh, and Paul Szymkowiak.

Feb02 SASQAG

37

Attendees (Last updated December 30, 2000.)

James Bach, Satisfice (co-host)

Scott Chase, Florida Institute of Technology

Sam Guckenheimer, Rational (co-host)

Elisabeth Hendrickson, Quality Tree Consulting, late of Aveo

Ivar Jacobson, Rational

Cem Kaner, Florida Institute of Technology

Grant Larson, Rational

Brian Marick, testing.com (co-host)

Noel Nyman, Microsoft

Bret Pettichord, Pettichord.com

Johanna Rothman, Rothman Consulting Group (facilitator)

Keith Stobie, BEA

Paul Szymkowiak, Rational (co-host)

James Tierney, Microsoft

James Whittaker, Florida Institute of Technology

Helping from afar, but unable to attend

Robert V Binder, RBSC

David Delano, AGCS

Linda Rising

Architecture Achilles Heel

By Elisabeth Hendrickson & Grant Larson

Objective: Identify areas for *bug hunting* — relative to the architecture

Problem: This pattern addresses a problem defined by the following four questions:

- How can I use the architecture to evaluate the implementation of a system?
- How can I determine how to spend my time to discover the most severe bugs?
- Where can I spend my time to discover the significant risks affecting the system?
- How do you identify areas where bugs are most likely to live within a system?

Context: You are a tester and want to make best use of the testing time. Your goal is to find high severity bugs. You have a physical, functional, or dynamic map of the system in some form (or can create one with *Architecture Reverse Engineering).* You know what characteristics of the system are most important (e.g. is reliability more important than performance?). You may or may not be doing formal test planning.

Achilles Heel - Forces

Forces:

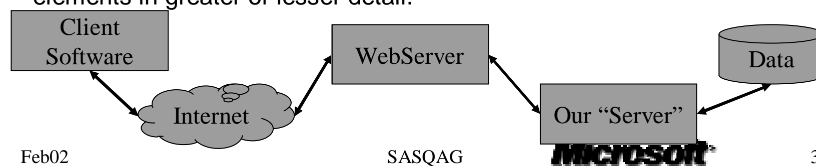
– You may not be able to answer all your own questions about the architecture.

– Getting information about the architecture or expected results may require additional effort.

– The goals of the architecture may not have been articulated or may be unknown.

Solution:

An architectural diagram is usually a collection of elements of the system, including executables, data, etc., and connections between those elements. The architecture may specify the connections between the elements in greater or lesser detail.



Achilles Heel - Walk Thru

Walk through the architectural diagram asking questions:

- For each executable m the architecture, what happens if that executable is not running?
- For each data storage mechanism (whether a relational database, file in the file system, or other persistent mechanism), what if the data is corrupted? Deleted? Lacks integrity or conflicts with other data in the system?
- For each connection, what if the connection breaks? Does it matter if the connection breaks for a short duration or a long duration? (In other words, are there timeout weaknesses?)
- Are there multiple connections going into or out of an element? Does this open up the possibility of resource conflicts or deadlocks?
- Are there shared resources where there may be performance bottlenecks?
- Do the elements of the architecture have states? Is there any possibility that those states might conflict—for example, could one part of the system try to start up a process while another part is trying to shut it down?

Achilles Heel - Rationale

As you answer the questions mentally or verbally, write down possible risks, weaknesses, or tests that occur to you. These become areas of the system on which to focus your energy. Alternatively, you could use the architectural diagram while exploring the system hands-on.

Where you are unable to answer your own questions, seek the assistance of others who can help you: programmers, architect(s), other testers, technical support personnel, etc.

Rationale:

Understanding the big picture of the system under test can help testers focus their energy appropriately. Without this understanding it's very easy to spend large amounts of time investigating relatively minor bugs or risks.

Achilles Heel - Result

Resulting Context:

You now know more about the architecture and implications of various design decisions and can use that information to guide your test effort. If the architectural goals are still unclear, perhaps an *Explicit Architectural Goal Articulation* is in order.

Additional Benefits:

- Increases communication between different groups within the project.
- Finds potential architectural flaws earlier.
- Aligns architecture with its goals.
- Provides additional fodder for test planning for current and future releases.

Achilles Heel - Liabilities

Liabilities:

- You may need to spend a large amount of time understanding the system before you can productively seek out bugs.
- Other members of the team or organization may need to invest more time answering questions to contribute to the test effort than anticipated.
- Some members of the team may feel threatened during this process.
- Information uncovered during this process may throw the project into chaos, at least temporarily.

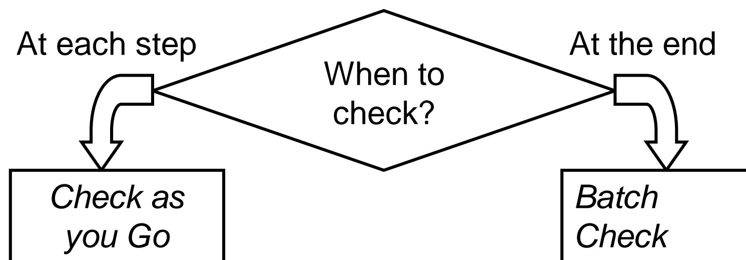
Related Patterns:

- Architecture Reverse Engineering
- Explicit Architectural Goal Articulation

Test Results

Check as you Go is the generally preferred method for ease of understanding.

Batch check (or *Benchmark*) is particularly useful for changing the *Judging* pattern into *Solved Example.*



Pattern Name: *Check as you Go*

Context

You have pre-specification of the test results

Do you compare results at each step as you go, that is to known expected results, or all at the end?

- Future results may be invalid if early results don't pass (see also *Separable Tests*)
- Relatively exact checks of the results are desired.
- The amount of output per result is relatively small

Forces

- Future results may be invalid if early results don't pass.
- Constantly interspersing checking code in testware can make the logic of a test case unclear.

Solution

Check each result **in-line** as finely as possible immediately after its inputs are submitted. Show the input and expected output in the same file.

This can include, for example, bounding the time of the result.

Check as you Go - Indications/Rationale

Indications

- The results of a test input are precisely known ahead of time.
- Programmatic inputs are used

Rationale

It generally aids comprehensibility of the tests if the expected results appear in the same file and as close to the inputs as possible.

Resulting Context/Consequences

Maintainability is sometimes reduced if bulk updates of results are needed.

Reduces code reusability if inputs and results are hard coded into the code.

Check as you Go - Uses & Patterns

Related Patterns

See *Batch check* for the same problem, but different contexts.

Examples/Known Use

Frequently used for API/Class Drivers approach.

POSIX Verification Test Suite.
Expect tool

A suite of code was developed to check the ULOG in real time to enhance this style of programming when one of the expected results is a ULOG message. Thus some of the many javaserver configuration tests check for a ULOG message explicitly as part of the test.

Check as you Go - Code Samples

Note below that the result is checked **in-line** in the code and not by some external entity.

Java/C++

```
result = squareRoot(1);
if (result != 1) {
LogError( "squareRoot(1) resulted in "+result
  +" where 1 was expected" )
}
```

Shell:

```
result=`squareRoot(1)`
if [ "$result" != "1" ] ; then
  echo "squareRoot(1) resulted in $result, where
  1 was expected."
  fi
```

Pattern Name: *Batch check*

Aliases:

Benchmark where a benchmark file containing expected results is used.

Context

You have pre-specification of the test results. The specification may be by hand judging actual results.

- set of inputs is not easily separable (e.g. compiler input file).
- output can be compared easily with minimal filtering.
- output is voluminous.

Problem

Do you compare results at each step as you go or all at the end?

Forces

- Avoiding false positives is more important than avoiding false negatives
- Future results may be invalid if early results don't pass.

Batch Check - Solution

Solution

Provide a benchmark file of expected results. Collect actual results as the test executes. At the end compare the expected and actual results.

Indications

- Output is difficult to predict and can frequently change from release to release (e.g. stub generation from CORBA IDL).
- A failure near the start of the test doesn't invalidate the results that follow.

Rationale

Very easy to develop. Expected results can be generated by the program once and hand checked for accuracy once and then reused again and again. This changes the *Judging* pattern into *Solved Example*. Expected results can be updated without effecting any code (since they are in a separate file). **Batch** processing may be the nature of item of test.

Batch Check - Consequences

Resulting Context/Consequences

One dangerous Consequence frequently seen is testers get lazy when the expected output has to change and *don't scrutinize* the initial results carefully enough for correctness. Then the actual, wrong, output gets canonized as the expected output.

Can make maintenance more difficult if the relationship between inputs and outputs is not very clear.

Frequently need special filtering patterns (regular expressions) to ignore uncontrollable extraneous differences, e.g. machine names, time stamps, etc. This filtering out has a small risk of missing incidental problems, like the time being reported wrong – generally you rely on other tests to specifically verify what most of these types of tests ignore.

BatchCheck - Uses & Patterns

Related Patterns

See *Check as you Go* as the generally preferred method.

Examples/Known Uses

Compiler testing or any transformation type program. It is generally too expensive to test each feature completely individually and a lot of common setup exists to test any one feature.

Code Samples

The abbreviated example below shows the expected output stored in a separate file and then a **batch** comparison done.

Shell:

```
echo 1 >expectedOutput

squareRoot(1) >> actualOutput
diff expectedOutput actualOutput
```

More PoST

- Quality Week, June 2001 talk by Keith Stobie
- PoST3 – Aug 26-27, Lexington, MA
- StarWest 2001, Workshop by Brian Marick

ChiliPLoP: **Feb. 25-28, 2002**, Carefree, AZ
http://www.agcs.com/supportv2/techpapers/patterns/chiliplop/2002/2002_testpatterns.htm

Test Pattern repository:
http://testing.com/test-patterns/

Microsoft®

This presentation is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.
© 2001 Microsoft Corporation. All rights reserved.