Pattern language for testing object-oriented software

Object-oriented patterns will be viewed as the single most important advance of the 1990s

Donald G Firesmith

ALTHOUGH THERE IS always a considerable risk associated with making predictions, I am convinced that object-oriented patterns will be viewed as the single most important advance made by the object community during the 1990s. As evidence for this prediction, consider that within a single year of its publication, the book DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE[1] has been recognized as the number one book in the list of the top ten all-time classics in the object technology field.[2]

According to the DICTIONARY OF OBJECT TECHNOLOGY,[3] a *pattern* is "a reusable architecture that experience has shown to solve a common problem in a specific context."* Patterns increase reuse and greatly improve communication among developers by providing a standard terminology.

Related patterns should be gathered together to form a pattern language designed to deal with a specific problem area such as user interfaces, database wrappers, telecommunications, banking, etc. Although patterns and pattern languages have been documented for many different domains, until recently, testing object-oriented software has received little notice. Few realize just how many articles and conference papers have been produced in the field. These sources, as well as over a decade of experience and industry contacts, has allowed me to recognize and document those patterns most commonly used in testing.

This article introduces the Pattern Language for Object-Oriented Testing (PLOOT), the first† pattern language to collect patterns con-

* A *context* is a general reason for using a pattern.

cerned with the testing of object-oriented software. PLOOT has been developed specifically to deal with the following general issues that differentiate object-oriented testing from traditional testing:

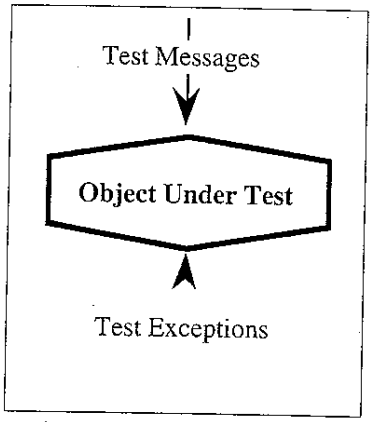


Figure 1. Class via Instance pattern.

- Objects are essentially software blackboxes that hide their encapsulated properties (eg, attributes, links, component parts, exceptions) and hidden operations behind an interface of visible operations.
- Individual operations within a class cannot be tested in isolation because they depend on their environment in the class (eg, other operations, properties, exceptions, assertions).
- Objects may exhibit externally recognizable state behavior captured in the form of equivalence classes of property values.
- Objects collaborate via messages and exceptions, whereby messages may be dynamically bound to polymorphic operations. Rather than test data sent to a function, a unit test case is therefore typically a:

  1. test message (with or without arguments) sent to an object under test that is in a specific state.
  2. test exception (with or without specific property values) raised to an object under test that is in a specific state.‡

- Objects are defined by means of classes that are related via inheritance relationships.
- Integration testing must take into account collaboration, inheritance, aggregation, and attribution as well as design patterns and mechanisms.
- An iterative, incremental, parallel development cycle is often used in which testing is performed incrementally, must be repeated often, and is begun earlier in the project schedule.

A *force* is anything that determines whether a pattern is applicable in a given context and influences the form of the solution provided by the pattern. The primary forces influencing the testing of object-oriented software are abstraction, inheritance, encapsulation, coupling, simplicity, uniformity across languages, consistency between software to be tested and test software, timeliness of fault identification, dependency, and the impedance mismatch of using multiple paradigms.

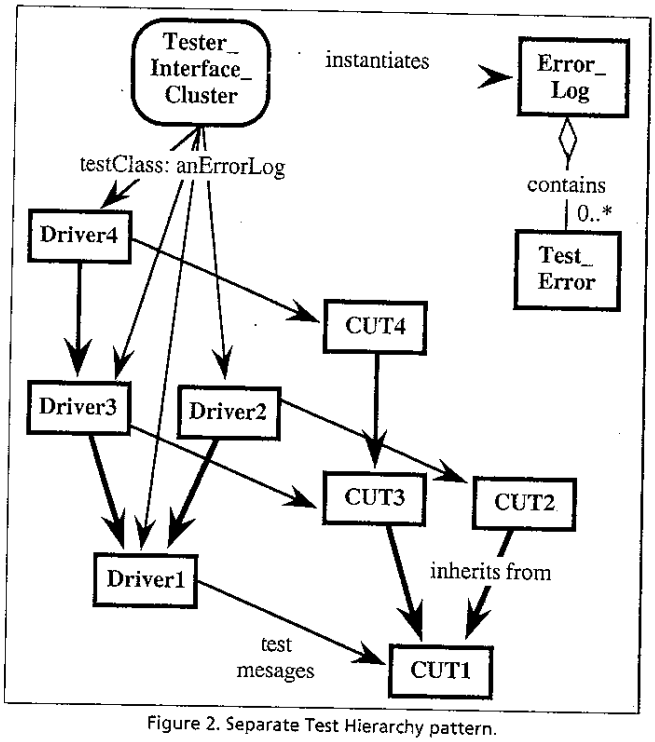


Figure 2. Separate Test Hierarchy pattern.

The testing patterns in the PLOOT pattern language can be grouped according to the following *contexts*:

- Unit tests:

  1. CLASS AS UNIT pattern. *Problem*: what is the unit of unit test? *Positive forces*: abstraction, encapsulation, inheritance, and uniformity. *Solution*: the class is the unit of unit test.
  2. CLASS VIA INSTANCE pattern. *Problem*: if classes are the unit of unit test, then how are classes to be tested when they may not be executable? *Positive forces*: uniformity, only objects are executable.§ *Solution*: indirectly test the class by testing one of its instances.
  3. TEST MESSAGES AND EXCEPTIONS pattern. *Problem*: if objects are the things that are actually tested, how are they tested? *Forces*: encapsulation, collaboration, reliability, robustness, and uniformity. *Solution*: each test is either a message sent to the object under test (OUT) or an exception raised to the OUT (see Fig 1).
  4. ASSERTIONS AND EXCEPTIONS pattern. *Problem*: if objects are the things that are actually tested, how are they tested? *Positive forces*: encapsulation, timeliness, reliability, and robustness. *Negative force*: uniformity. *Solution*: embed assertions (ie, class invariants and operation preconditions and postconditions) in the code, and raise exceptions when they are violated.

- Test case form:

  5. TEST CASE AS OPERATION pattern. *Problem*: how to store simple test cases. *Positive force*: uniformity. *Negative forces*:

† Anuradha Kare and John McGregor of Clemson University are also in the process of developing a testing pattern language/framework, tentatively called PACT.

‡ Additional messages may be used to place the object under test into the appropriate pre-test state, verify that it is in the appropriate pretest state, determine whether the test case places it in the appropriate post-test state, and determine whether it sends the appropriate messages to its collaborators.

§ A class can be directly tested if it has class-level properties and behavior. *Abstract* classes can be subclassed to supply deferred characteristics and then instantiated. *Generic* classes can be supplied with actual parameters and then instantiated.

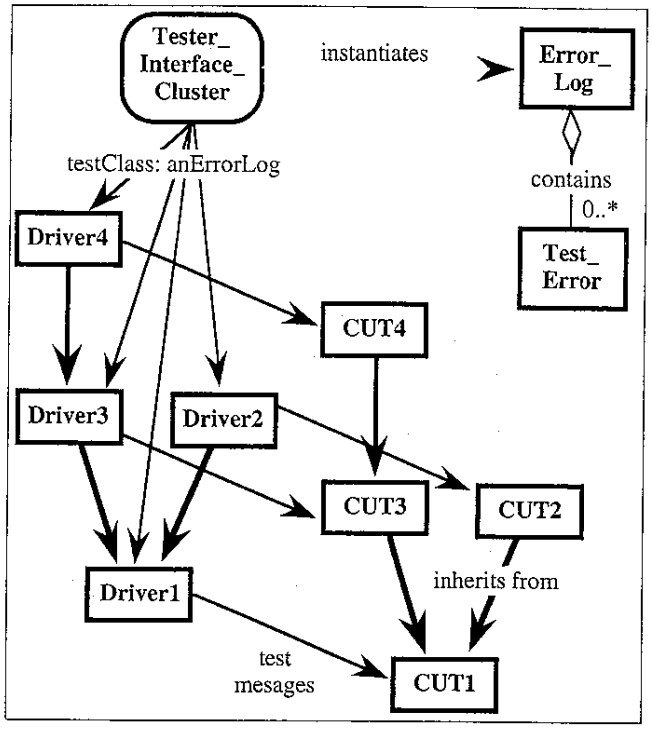


Figure 3. Mixin and Join Test Hierarchies pattern.

abstraction and encapsulation. *Solution*: store each test case as an operation that sends the test message or raises the test exception. Store the parameters and oracle in local variables.

6. TEST CASE AS OBJECT pattern. *Problem*: how to reify test cases so that the complex test cases can be reviewed, updated, and displayed. *Positive forces*: abstraction, encapsulation, and uniformity. *Solution*: store each test case as an object that can be queried, updated, and executed. Store the test name, test purpose, test message parameters, and expected test results as attributes of the test object.

- Test script form:

7. TEST SCRIPT AS OPERATION pattern. *Problem*: how to store simple test scripts containing multiple test cases. *Positive force*: uniformity. *Negative forces*: abstraction and encapsulation. *Solution*: store each test script as an operation that executes the associated test cases.

8. TEST SCRIPT AS OBJECT pattern. *Problem*: how to reify test scripts. *Positive forces*: abstraction, encapsulation, and uniformity. *Solution*: store each test script as a collection object that can be queried and updated, and that can execute the associated test case objects.

- Class test case design:

9. EVERY EVENT IN EVERY STATE pattern. *Problem*: how to identify white-box state-based test cases. *Positive forces*: abstraction, inheritance, and uniformity. *Solution*: instantiate the class under test to produce an object under test. For every state of the OUT, compare its actual response to its expected response to every kind of test event (ie, test message or test exception). The response includes the return value of the message, the OUT's post-test state, and any resulting messages delegated to collaborators of the OUT.[4]

10. PAIRED TEST SCENARIOS pattern. *Problem*: how to identify black-box state-based test cases. *Positive forces*: abstraction and uniformity. *Solution*: use the specification or state model of the class under test to develop a set of pairs of test scenarios (ie, sequences of messages). For each pair of test scenarios, determine if the application of the test scenarios would result in equal objects if applied to two newly instantiated instances of the class under test. Instantiate the class under test twice to produce two equal objects under test and apply each test scenario to one of the objects under test. Determine if the two resulting are in fact equal.[5]

11. TRANSITION TREES pattern. *Problem*: how to identify black-box state-based test cases from a state transition diagram. *Positive forces*: abstraction and uniformity. *Solution*: use the state model of the class under test and the method described in Chow[6] to produce a transition tree. The nodes on the transition tree represent states of instances of the class under test, and the edges on the transition tree represent state transitions. The root node on the tree represents the initial state of the instance. For each leaf node state of the tree and each transition from that state, draw an edge representing the corresponding message to the new leaf node representing the corresponding new state. Each test case then consists of a sequence of test messages. The first test message in a sequence instantiates an object under test in the proper initial state. The last test message returns the final state of the object under test. The other test messages are defined by the sequence of edges of a complete or partial branch of the transition tree starting at its root node (ie, initial state).[7]

- Class test case location:

12. BUILT-IN TESTS pattern. *Problem*: how to bypass encapsulation and inherit test cases. *Positive forces*: encapsulation, inheritance, coupling, simplicity, uniformity, and consistency. *Solution*: include a testClass class-level operation and a testInstance object-level operation in each class. For each operation, include an associated protected object-level operation that applies the associated test cases to that operation.[4,8,9]

13. SEPARATE TEST DRIVER HIERARCHY pattern. *Problem*: how to test an operation that delegates in terms of expected messages to a collaborator and exceptions raised by a collaborator. *Positive forces*: abstraction, encapsulation, and inheritance. *Negative forces*: coupling, simplicity, uniformity, and consistency. *Solution*: Create a separate hierarchy of test drivers whereby each test driver sends test messages to the corresponding class under test. This pattern often involves significantly more complex test cases because it is more difficult to place the class under test into the appropriate pre-test state, verify it is in the correct pre-test state, and verify it is in the expected post-test state (see Fig 2).[10]

14. TEST SUBCLASSES pattern. *Problem*: how to bypass encapsulation and inherit test cases, while keeping the test cases separate from the classes under test. *Positive forces*: inheritance and encapsulation. *Negative forces*: coupling, simplicity, uniformity, and consistency. *Solution*: subclass each class under test, including all test operations and test cases for the class under test. Indirectly test the class under test by testing its test subclass. This pattern exists in two variants: either the test subclasses do not inherit from each other, in which case significant test software must be copied manually, or else the test subclasses multiply inherit from both the class under test and the test subclass(es) of the parent class(es) of the class under test, in which case shared inheritance rather than repeated inheritance is required.

15. MIXIN AND JOIN TEST HIERARCHIES pattern. *Problem*: how to bypass encapsulation and inherit test cases, while keeping the test cases separate from the classes under test. *Positive forces*: inheritance and encapsulation. *Negative forces*: coupling, simplicity, uniformity, and consistency. *Solution*: develop a hierarchy of mixin classes,# each of which corresponds to a class in a hierarchy of classes to be tested. Each mixin class contains (either directly or via inheritance) the test operations and test cases needed to test the corresponding class under test. Using multiple inheritance, create a join class for each class under test and its corresponding mixin class. Indirectly test each class under test by testing the corresponding join class (see Fig 3).

16. VOYEUR pattern.‖ *Problem*: how to bypass encapsulation and inherit test cases, while keeping the test cases separate from the classes under test. *Positive forces*: inheritance and encapsulation. *Negative forces*: coupling, simplicity, uniformity, and consistency. *Solution*: for each hierarchy of classes to be tested, develop a corresponding hierarchy of voyeur classes, each of which contains the test operations and test cases required to access and test the private parts of the corresponding class under test. Implement the voyeur classes using friends in C++, extensions in ENVY Smalltalk, or child units in Ada95.

Although patterns and pattern languages have been documented for many different domains, until recently, testing object-oriented software has received little notice.

A mixin class is an abstract class containing features (properties or operations) intended to be mixed via multiple inheritance with the features of another class to form a join class.

‖ The name of this pattern comes from its roots in C++, the language in which you can allow your "friends to view your private parts."

- Acceptance testing: Acceptance testing is supported by the following patterns:

17. USE CASE TESTING pattern. *Problem*: how to test reactive systems with a user interface. *Positive force*: user orientation. *Negative forces*: the numerous document potential problems associated with use cases.[9] *Solution*: for each use case, use the associated usage scenarios as acceptance test cases.

18. RECORD AND PLAYBACK pattern. *Problem*: how to regression test a graphical user interface (GUI). *Positive force*: existence of commercial tools. *Solution*: use an object-aware record and playback tool to record the correct interaction between the user and the GUI, and then replay the user actions after a change has been made and compare the resulting GUI actions with its previously recorded actions.

- Test infrastructure:

19. TEST HARNESS pattern. *Problem*: how to provide a standard tester interface. *Positive forces*: reuse and consistency among developers. *Solution*: create a standard test harness that allows the tester to browse through the library of classes to test, send the testClass message to the class under test, and capture the list of test error objects returned.

20. TEST STUBS pattern. *Problem*: how to test an operation that delegates in terms of expected messages to a collaborator and exceptions raised by a collaborator. *Negative forces*: encapsulation and coupling. *Solution*: for each collaborator of the object under test, create a test stub that records all messages sent by the object under test as a result of a test message and raises test exceptions to the object under test.

21. TEST FRAMEWORKS pattern. *Problem*: how to combine and implement the preceding patterns. *Positive forces*: cost and consistency. *Solution*: implement a framework that includes a consistent subset of the patterns in this language. For example, include within the framework patterns 1–4, 6, 8–9, 12, and 17–21, and use this pattern in accordance with patterns 22–25.

- Test process:

22. INCREMENTAL, ITERATIVE, PARALLEL TESTING pattern. *Problem*: how to test consistently with the object-oriented development cycles. *Positive forces*: consistency with the development cycle. *Solution*: perform testing in an incremental, iterative manner according to the tenet "analyze a little, design a little, test a little."

23. AUTOMATED REGRESSION TESTING pattern. *Problem*: how to minimize the increased regression testing. *Positive forces*: inheritance, reuse, and consistency with development cycle. *Solution*: use the TEST FRAMEWORKS pattern to easily automate regression testing.

24. EARLY TESTING pattern. *Problem*: how to test to minimize cost and maximize bug fixing. *Positive forces*: cost and quality. *Solution*: test each component as soon as is practi-

cal in accordance with the project's object-oriented development cycle.

25. DEPENDENCY-BASED TESTING pattern. *Problem*: how to determine an optimum test order. *Positive forces*: effort, cost, and testability. *Solution*: test the classes in dependency order based on inheritance, delegation (ie, message passing), aggregation, and attribution, starting with classes that do not depend on any other classes, and then continuing with classes that only depend on previously tested classes.

CONCLUSION

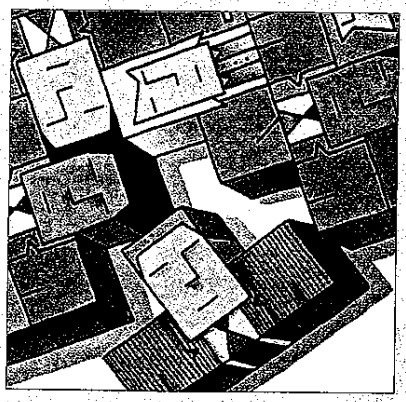
In summary, this article has very briefly introduced 25 patterns that have been shown by experience to provide standard solutions to most of the problems associated with testing object-oriented software. Because of space limitations, these patterns will be documented in more detail in a series of upcoming columns in the REPORT ON OBJECT ANALYSIS AND DESIGN and form a large chapter in my forthcoming book TESTING OBJECT-ORIENTED SOFTWARE.

References

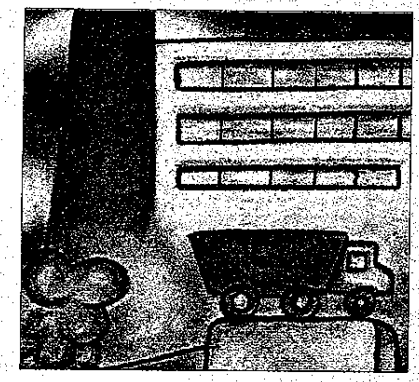
1. E Gamma et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.
2. S Bilow. Five for '95: This year's hot new books, JOURNAL OF OBJECT-ORIENTED PROGRAMMING 8(5):43–46, SIGS Publications, New York, 1995.
3. D Firesmith and EM Eykholt. DICTIONARY OF OBJECT TECHNOLOGY: THE DEFINITIVE DESK REFERENCE, SIGS Publications, New York, 1995.
4. D Firesmith. Object-oriented regression testing, REPORT ON OBJECT-ORIENTED ANALYSIS AND DESIGN 1(5):42–45, 1995.
5. P Frankl and R Doong. Tools for testing object-oriented programs, PROCEEDINGS OF THE 8TH ANNUAL SOFTWARE QUALITY CONFERENCE, Pacific Agenda, Portland, OR, 1990, pp. 309–325.
6. T Chow. Testing software design modeled by finite state machines, IEEE TRANSACTIONS 4(3):178–186.
7. R Binder. State-based testing, OBJECT MAGAZINE 5(4):75–78, SIGS Publications, New York, 1995.
8. D Firesmith. Testing object-oriented software, PROCEEDINGS OF TOOLS USA '93), R Ege, M Singh, and B Meyers, eds, Prentice Hall, Englewood Cliffs, NJ, 1993, pp. 407–426.
9. D Firesmith. Use case: the pros and cons, REPORT ON OBJECT-ORIENTED ANALYSIS AND DESIGN 2(2):2–6, 1995.
10. J McGregor. Testing object-oriented software systems, (tutorial), OOPLSA '95, Austin, Texas, October 1995.

Donald G Firesmith is Senior Member of Technical Staff at Knowledge Systems Corp, Cary, NC.

Finding and using patterns 24



Using patterns to teach design 40



Using use cases 50

OBJECT magazine

A SIGS Publication

PATTERNS

23 Focus on patterns: An introduction

24 Patterns 101 — Kent Beck

What exactly are patterns? How do you define them, find them, and use them to codify software development?

32 Pattern language for testing object-oriented software — Don Firesmith

Here are 25 patterns that have been shown by experience to provide standard solutions to most of the problems associated with testing object-oriented software.

40 Experiencing patterns at the design level — Kyle Brown

Teaching novices how to become good OO designers is one of the toughest problems a person can face. With the advent of design patterns, we may now have a medium to make some of those lessons easier.

USE CASES

50 Teaching model-based software engineering — Robert F Coyne & Allen H Dutoit

In this software engineering course, Objectory was used to provide a realistic experience by immersing students in a single, team-based, system design project to build and deliver a complex system for a real client.

QUALITY

56 Software quality begins with a quality design — Stephen Kaufer & Webb Stacy

Achieving software quality takes two steps: produce a correct specification and then translate it into a functionally correct product. Completing only one step doesn't get you very far.

61 Breaking OO out of the "Time Box" — David Linthicum

Supporting the promise of RAD is a new array of 4GL GUI wonder-tools that allow developers to assemble systems as quickly as General Motors assembles cars.

Table of Contents

5(8) January 1996

DATABASES

64 **Smalltalking to a relational database** — Bolette Andersen, Barbara Miller, David Pitts, & Jay Johnson

Making an object model fit into a relational schema can be akin to forcingthe proverbial round peg. In the authors, case, however, there was a ray of hope: objects in the application could be accessed by means of special objects known as "keys."

COLUMNS

16 **Methodology:** Process objects: Back from programming "Siberia" — John Palmer

20 **Leadership:** Objects alone are not sufficient — Lynn Chilson

71 **Architectures:** Kick the stovepipe systems habit — Tom Mowbray

72 **Databases:** Object relational integration technologies — Joshua Duhl

75 **Distributed Objects:** Understanding the CORBAservices Naming Service — Mark Roy & Alan Ewald

78 **Migration Strategies:** Code reuse doesn't work — Ian Graham

96 **Education & Training:** Patterns for pedagogy — Susan Lilly

DEPARTMENTS

4 **From the Editor:** Productivity tools — Marie A Lenzi

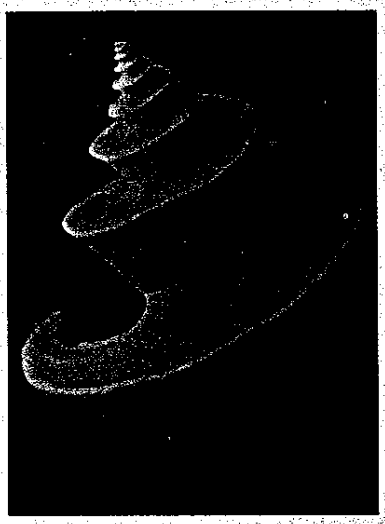
8 **Executive Brief** — Michael Kei Stewart

12 **Industry Analysis:** Object technology markets in review — Adrian Bowles

80 **Ad Index**

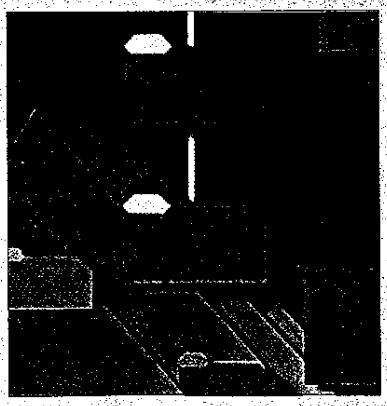
81 **Product Track** — Jane M Grau

87 **Book Watch** — Jane M Grau



Cover photograph by Yoichi Nagata

56 2 keys to software quality



64 Smalltalking to an RDBMS