The PLOOT Test Pattern Language

Donald G. Firesmith

Donald G. Firesmith is a senior member of the technical staff at Knowledge Systems Corporation in Cary, NC. He may be contacted at dfiresmith@ksccary or 73664.3513@compuserve.com.

THE PLOOT PATTERN LANGUAGE

Although there is always 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[1] was recognized as the number one book in the list of top-10 all-time classics in the object technology field.[2]

According to the DICTIONARY OF OBJECT TECHNOLOGY,[3] a *pattern* is "any reusable architecture that experience has shown to solve a common problem in a specific context." A pattern language is a consistent collection of related patterns gathered together to deal with a specific problem area (e.g., user interfaces, database access, telecommunications). Both individual patterns and pattern languages are used to capture and distill reusable knowledge about a domain, make it easier to reuse successful approaches, choose among alternative approaches based on their different applicabilities, enhance communication among developers by providing a common term for common designs, improve documentation, and improve the quality of software developed.

The *Pattern Language for Object-Oriented Testing* (PLOOT) is the first published pattern language created for the testing of object-oriented software.[4] The patterns in PLOOT are briefly summarized as follows, organized both by context and by the dependency relationships among* them:

CONTEXT CLASS/CLUSTER TESTING:

Pattern 1. Class as Unit (CAU)

Problem: What is the unit of unit testing?

Solution: Classes.

Pattern 2. Concrete Class via Instance (CCVI)

Problem: How do you test concrete classes when they are primarily specifications of objects and are therefore often not executable?

Solution: In accordance with the CAU pattern, primarily test concrete classes indirectly via their instances.

Pattern 3. Abstract Class via Subclass (ACVS)

Problem: How do you test abstract classes when they either cannot or should not be instantiated?

Solution: In accordance with the CAU and CCVI patterns, primarily test abstract classes indirectly via instances of concrete test subclasses.

Pattern 4. Test Messages and Exceptions (TM&E)

Problem: What are the test inputs?

Solution: Send test messages and raise test exceptions.

Pattern 5. Test Stubs (TSTUB)

Problem How do you test objects/classes that delegate some of their responsibilities to other objects?

Solution: Use test stubs that (1) verify delegation by the object under test (OUT) and that (2) raise test exceptions to the OUT.

Pattern 6. Dependency-Based Testing (DBT)

Problem: What is the optimum order in which to test classes so as to minimize the test effort and maximize the testability of the classes tested?

Solution: Develop a dependency diagram that documents the various dependency relationships from client to server among the classes including inheritance, aggregation, collaboration/delegation, attribution, and parameterization. To test new classes on a foundation of previously tested classes, unit/integration test all classes on the dependency diagram starting with the root(s) of the dependency diagram and working toward the leaves so that no class is tested before its dependents are tested. If two classes depend on each other, stub out the more complex class and test the simpler class first. Also, test the methods in a class in dependency order; typically (1) read accessors before write accessors before constructors before queries before ser-

* Where practical, patterns that depend on other patterns are listed after those patterns.

vices and (2) private operations before public operations.

Pattern 7. Cluster via Classes (CVC)

Problem: How should you test a cluster of collaborating classes in terms of its component classes?

Solution: In accordance with the DBT pattern, perform as much of the cluster integration testing during the bottom-up dependency-based testing of its component classes.

Pattern 8. Cluster via Whitebox Scenarios (CVWS)

Problem: How should you test a cluster of collaborating classes in terms of the scenarios that thread through it?

Solution: For each use case using a part of the cluster, determine the primary and secondary whitebox (partial) scenarios involving the classes in the cluster. In accordance with the CCVI, ACVI, and the TM&E patterns, test each whitebox scenario with an associated test case using associated test drivers to provide inputs to the cluster and monitor actual returned values and raised exceptions and associated test stubs that monitor the actual delegations. Compare expected with actual returned values, raised exceptions, and delegations.

Pattern 9. Assertions and Exceptions (A&E)

Problem: How do you identify the fault that caused a failure as it happens so that appropriate actions (e.g., debugging or failure recovery) can be performed?

Solution: Where practical, embed appropriate assertions (i.e., class invariants, operation preconditions and postcondition), exceptions, and exceptions handlers in the classes, check these assertions at run-time, raise exceptions when they are violated, and perform appropriate actions where the exceptions are caught.

CONTEXT TEST CASE DESIGN:

Pattern 10. Every Event in Every State (EEES)

Problem: How do you create a relatively complete set of *whitebox* test cases when the instances of the class under test have significant encapsulated (state) properties?

Solution: Use the class state transition diagram to create test cases that send every kind of test event (i.e., a representative of each equivalence class† of test message with arguments or test exceptions with properties) to the associated object under test (OUT) in every one of its states. Compare the OUT's actual responses (e.g., return value or raised exception, post-test state and property values, delegated messages) to the expected ones provided by the oracle.

Pattern 11. Transition Trees (TT)

Problem: How do you create a relatively complete set of *blackbox* test cases when the instances of the class under test have encapsulated state properties?

Solution: Use the class state transition diagram to create a transition tree. Create test cases of test message scenarios‡ that correspond to partial branches of the transition tree.§ Execute the test cases, starting at the root of the transition tree and working to the leaves. Compare the OUT's actual responses (e.g., return value or raised exception and post-test state and property values) to the expected ones provided by the oracle.

Pattern 12. Paired Test Scenarios (PTS)

Problem: How do you create a set of *blackbox* test cases when the instances of the class under test either is specified via axioms or a state transition diagram?

Solution: Based on either the axioms or state transition diagram, create pairs of test message scenarios. Send the two test message scenarios to two newly constructed equal instances of the class under test, determine if the resulting objects are equal, and compare this actual result with the result that was expected based on the axioms or state transition diagram.

† If the design is correct, each representative should elicit the same qualitative behavior from the class under test.

‡ A test message scenario is a sequence of related test messages sent to the same object under test.

§ A partial branch includes all states and transitions from the starting state to the root state of the tree.

CONTEXT TEST CASE, SCRIPT, AND SET FORM:

Pattern 13. Test Cases, Scripts, and Sets as Operations (TCSSOP)

Problem: How do you store simple, stable test cases, scripts, and sets?

Solution: Store the test set of each class as a test set operation that executes the test script operations that in turn execute the test case operations that send test messages and/or raise test exceptions to the objects or classes under test.

Pattern 14: Test Cases, Scripts, and Sets as Objects (TCSSOB)

Problem: How do you store complex test cases, test scripts, and test sets with associated information that must be accessed and maintained as well as executed? For example, a test case should store its purpose, test message parameters, test exceptions including properties, required pre-test state, and oracle including expected return value or exception, post-test state, and delegations to collaborators?

Solution: Reify such test cases as test case objects that send test messages and/or raise test exceptions. Reify such test scripts as collection objects that contain the associated test case objects as component parts. Provide the necessary iterator and accessor operations so that the encapsulated test cases can be executed and the test script's attributes can be accessed and maintained. Similarly reify the test sets as collections of test scripts.

CONTEXT TEST CASE, SCRIPT, AND SET LOCATION:

Pattern 15. Built-in Tests (BIT)

Problem: How do you store *whitebox* class test cases, scripts, and sets so that they may be inherited and address problems of encapsulation?

Solution: Encapsulate new and overridden protected test cases, scripts, and sets in the associated classes under test. Add standard test operations to the protocols of the root classes under test so that the test scripts and test cases can be executed from the outside without violating their encapsulation.

Pattern 16. Separate Test Driver Hierarchy (STDH)

Problem: How do you store *blackbox*

class test cases, scripts, and sets so that they may be inherited and easily separated from the deliverable classes under test?

Solution: Create a separated inheritance hierarchy of test drivers, each of which contains the test cases, scripts, and sets for the associated class under test (CUT). Have each test case send the appropriate test messages to the CUT.

Pattern 17. Test Subclasses (TSC)

Problem: How do you store *whitebox* class test cases, scripts, and sets so that they may be inherited and address problems of encapsulation?

Solution 1: For each class under test (CUT), create a child test subclass that encapsulates the corresponding new or overridden protected test cases, scripts, and sets and have the original child subclasses of the inherit from the new child test subclasses. Indirectly test each CUT by testing its corresponding test subclass. Add standard test operations to the protocols of the root classes under test so that the test scripts and test cases in the test subclasses can be executed from the outside without violating their encapsulation.

Solution 2: For each class under test (CUT), create a child test subclass that encapsulates the corresponding new or overridden protected test cases, scripts, and sets. Have the original child subclasses of the inherit from the new child test subclasses, and have the test subclasses multiply inherit from both their corresponding CUT and the test subclasses of the original parent classes of the CUT. Indirectly test each CUT by testing its corresponding test subclass. Add standard test operations to the protocols of the root classes under test so that the test scripts and test cases in the test subclasses can be executed from the outside without violating their encapsulation.

Pattern 18. Mixin and Join Test Hierarchies (MJTH)

Problem: How do you store *whitebox* class test cases, scripts, and sets so that they may be inherited, address problems of encapsulation, and be easily separated from the deliverable classes under test?

Solution: Create an analogous inheritance hierarchy of mixin test classes whereby each class in the mixin hierarchy encapsulates test cases, scripts, and sets that test the corresponding class under test (CUT) in the original inheritance hierarchy. Create a third hierarchy of join classes, each of which multiply inherits from the corresponding mixin class and CUT. Indirectly test the CUT by testing the corresponding join class.

Pattern 19. Voyeur Test Hierarchy (VTH)

Problem: How do you store *whitebox* class test cases, scripts, and sets so that they may be inherited, address problems of encapsulation, and be easily separated from the deliverable classes under test.

Solution: Create an analogous inheritance hierarchy of voyeur test classes (e.g., C++ friends, Ada subunits, Smalltalk ENVY extensions), each of which encapsulates test cases, scripts, and sets that test the corresponding class under test (CUT) in the original hierarchy. Indirectly test the CUT by executing the test cases and test scripts in the corresponding voyeur class.

CONTEXT ACCEPTANCE TESTING:

Pattern 20. Record and Playback (R&P)

Problem: How do you regression test a graphical user interface?

Solution: For each test script, record the inputs and correct outputs. After each modification to the system, playback the inputs and compare the new outputs with the original (i.e., expected) outputs.

Pattern 21. Use Case Testing (UCT)

Problem: How do you validate a reactive build, release, or [sub]system in terms of the functional requirements allocated to it?

Solution: For each use case, determine one or more primary usage scenarios exercising the primary path(s) through the use case. Augment these usage scenarios with secondary usage scenarios that exercise alternative or exceptional paths through the use case. For each usage scenario, create a test script (with associated test drivers and stubs representing the use case's actors) the provides the necessary inputs and monitors the expected outputs.

CONTEXT TEST INFRASTRUCTURE

Pattern 22. Test Browsers (TB)

Problem: How do you support the easy building of test cases, scripts, and sets, especially when they have a recursively complex internal structure of test messages, test exceptions, required pretest property values of the object or class under test, and oracles (e.g., returned values, raised exceptions, expected delegations)?

Solution: Build test case, script, and set browsers (or enhance existing browsers) that allow the tester to recursively build complex objects.

Pattern 23. Tester Interface (TI)

Problem: How do you provide standard support for the execution and maintenance of test scripts and test cases and the recording of test faults.

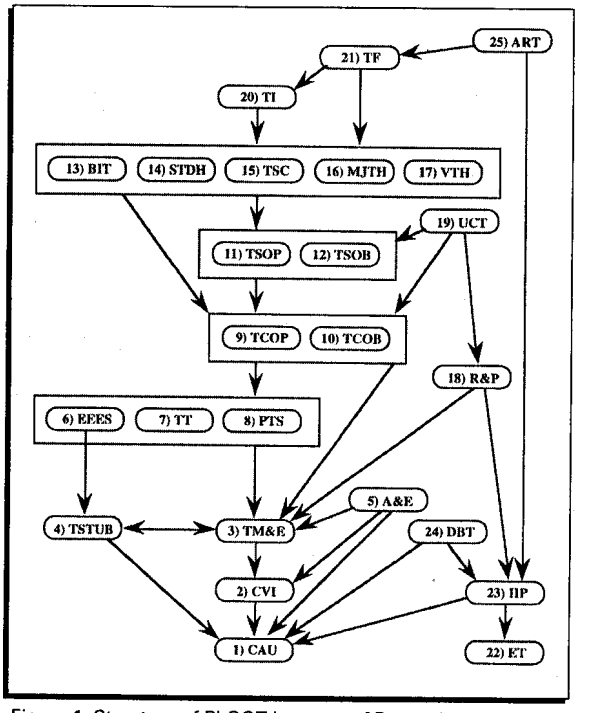

Figure 1. Structure of PLOOT in terms of Dependency Relationships.

Solution: Create a standard tester interface consisting of a cluster of classes that allows the tester to browse through the classes under test, execute and maintain test scripts and test cases, and log faults.

Pattern 24. Test Framework (TF)

Problem: How do you provide integrated tool support for the testing effort.

Solution: Build a test framework that implements a consistent set of the previous test patterns.

CONTEXT TEST PROCESS

Pattern 25. Iterative, Incremental, Parallel Testing (IIIP)

Problem: Where does testing fit into the object-oriented development cycle?

Solution: Perform testing as an integral part of an iterative, incremental, parallel development cycle. Perform regression testing as the subject of the test is iterated. Perform incremental testing as part of an "analyze a little, design a little, implement al little, test and integrate a little" process. If software is developed by teams working in parallel, then testing should also occur in parallel.

Pattern 26. Tester Responsibilities (TR)

Problem: How should test responsibilities be assigned to testers in order to maximize efficiency given human nature?

Solution: Have class developers perform and peer-review initial class/cluster testing. Have members of the independent test team complete class/cluster testing and perform acceptance testing. Have quality assurance monitor testing and regression test a statistical sample of the tests performed by the developers and independent testers.

Pattern 27. Automated Regression Testing (ART)

Problem: How do you minimize the regression test effort when classes need to be retested much more often than traditional functions due to reuse, inheritance, and iterative development?

Solution: Automate as much of the regression testing effort as is practical using the previous patterns.

PLOOT is summarized in Table 1, and the structure of the PLOOT pattern language is given by Fig. 1, which documents dependency relationships among the PLOOT patterns. The boxes represent groups of patterns related by context (e.g., patterns 15–19) any one of which can be used as part of a test framework.

CONCLUSION

In this column I have summarized the latest version of the PLOOT test pattern language. For the next few months, I will be using my future ROAD columns to document the individual PLOOT patterns in significantly more detail.

The material in this column is excerpted from my forthcoming book, TESTING OBJECT-ORIENTED SOFTWARE and printed here with the permission of SIGS Books.

References

1. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. DESIGN PATTERNS, Addison-Wesley, Reading, MA, 1995.
2. Bilow, S.G. Five for '95: This year's hot new books, JOOP 8(5):43–45, 1995.
3. Firesmith, D.G. and E.M. Eykholt. DICTIONARY OF OBJECT TECHNOLOGY. SIGS Books, New York, 1995.
4. Firesmith, D.G. Pattern language for testing object-oriented software. OBJECT MAGAZINE 5(8):32–38, 1996.

The Patterns in PLOOT

Pattern	Context	Problem Solved
1. Class as Unit	Class/Cluster testing	What is the unit of unit test?
2. Concrete Class via Instance	Class/Cluster testing	How to test concrete classes?
3. Abstract Class via Subclass	Class/Cluster testing	How to test abstract classes?
4. Test Messages and Exceptions	Class/Cluster testing	What are the test inputs?
5. Test Stubs	Class/Cluster testing	To test an operation that delegates
6. Dependency-Based Testing	Class/Cluster testing	Optimal test order?
7. Cluster via Classes	Class/Cluster testing	How to test clusters?
8. Cluster via Whitebox Scenarios	Class/Cluster testing	How to test clusters?
9. Assertions and Exceptions	Class/Cluster testing	How can objects self-test?
10. Every Event in Every State	Test Case Design	To identify whitebox test cases
11. Transition Trees	Test Case Design	To identify blackbox test cases
12. Paired Test Scenarios	Test Case Design	To identify blackbox test cases
13. Text Cases, Scripts, and Sets as Operations	Test Case, Script, and Set Form	How to store simple test cases, scripts, and sets?
14. Test Cases, Scripts, and Sets as Objects	Test Case, Script, and Set Form	How to store and maintain complex test cases, scripts, and sets?
15. Built-in Tests	Test Case, Script, and Set Location	How to bypass encapsulation?
16. Separate Text Driver Hierarchy	Test Case, Script, and Set Location	How to separate out test software?
17. Test Subclasses	Test Case, Script, and Set Location	How to bypass encapsulation?
18. Mixin and Join Test Hierarchies	Test Case, Script, and Set Location	How to bypass encapsulation?
19. Voyeur Test Hierarchy	Test Case, Script, and Set Location	How to bypass encapsulation?
20. Record and Playback	Acceptance testing	How to regression test a GUI?
21. Use Case Testing	Acceptance testing	To test a reactive system with UI
22. Test Browsers	Test Infrastructure	To test build and maintain complex test cases, scripts, and sets
23. Tester Interface	Test Infrastructure	To provide a standard tester API
24. Test Frameworks	Test Infrastructure	To provide a test framework
25. IIP Testing	Test process	To test consistently with lifecycle
26. Tester Responsibilities	Test process	How to assign test responsibilities
27. Automated Regressing Testing	Test process	To support regression testing