

Testing During Refactoring: Adding Aspects to Legacy Systems

Michael Mortensen
Hewlett-Packard
3404 E. Harmony Road, MS 88
Fort Collins, CO 80528

Sudipto Ghosh, James M. Bieman
Computer Science Department
Colorado State University,
Fort Collins CO 80523, USA

Abstract

Moving program code that implements cross-cutting concerns into aspects can improve the maintainability of legacy systems. This kind of refactoring, called aspectualization, can also introduce faults into a system. A test driven approach can identify these faults during the refactoring process so that they can be removed. We perform systematic testing as we aspectualize commercial VLSI CAD applications. The process of refactoring these applications revealed the kinds of faults that can arise during aspectualization, and helped us to develop techniques to reduce their occurrences.

Keywords: *Aspect-oriented programming, refactoring, legacy systems, test driven development, regression testing, mock system, test coverage*

1. Introduction

Refactoring systems by replacing cross-cutting code with aspects can improve design structure [16]. The use of aspects in the refactored *aspectualized* system helps modularize cross-cutting code concerns, thereby enhancing understandability and maintainability. As with any refactoring technique, a systematic approach to testing is needed to ensure that new faults are not introduced.

In an aspect-oriented program, cross-cutting concern are modularized into *aspects*. The aspects are *woven* into the primary code by a preprocessor, compiler, or run-time system. An aspect includes *advice*, which is the functionality to be woven in, and a *point-cut*, which identifies the locations in the primary code, called *joinpoints*, where the advice is inserted. Advice specified when it executes: *before*, *after*, or *around*. The *around advice* executes instead of the joinpoint but can execute the original joinpoint. Aspect languages such as AspectJ and AspectC++ offer a set of pattern-matching mechanisms to match advice to joinpoints. We use AspectC++ in this work, since we are refactoring a legacy C++ system.

Our approach involves (1) identifying cross-cutting code concerns and creating aspects, and (2) removing cross-cutting code from the system and weaving in the aspects. We are investigating the use of aspects in legacy systems to improve modularity of scattered code, to enable fast debugging, and to automatically enforce design policies.

Legacy systems present a challenge because of their large size. During the creation of aspects, it is difficult to experiment with changes in aspect pointcuts and advice due to long compilation and weave times. We use *mock systems*, “stub” systems that are orders of magnitude smaller than the real application, to develop and validate aspect pointcuts and advice before weaving them into the real application. Using the mock system serves as a unit testing phase in which we (1) identify essential characteristics of the real application to create the mock system, (2) analyze the weaving process to evaluate the use of aspects, and (3) apply test coverage criteria defined in terms of the connections between the aspects and the mock system.

In the integration testing phase, we test the refactored legacy system to ensure that the original functionality has not been broken. Again, we analyze the weaving process and apply coverage criteria to ensure that the aspects are woven correctly.

We developed a suite of tools to support weave analyses, test execution, and coverage measurement. We used our approach to refactor two VLSI CAD applications. The weave analysis and testing process detected a number of faults that were introduced during aspect creation and aspect integration. This paper describes our test approach, the faults that we detected, and potential faults that can occur.

In Section 2 we describe the applications being refactored and some identified aspects. We explain our mock-based unit testing and integration testing approach in Section 3. We examine faults and potential faults found during unit testing in Section 4, and those found during integration in Section 5. Related work is discussed in Section 6, followed by our conclusions and directions for future work in Section 7.

2. Aspectualizing VLSI CAD Applications

We are refactoring VLSI CAD applications with AspectC++ [14]. These applications are based on an object-oriented framework developed at Hewlett-Packard. The refactored applications use a newly-created aspect-library and continue to use the framework. This paper reports our work with two framework-based applications: PowerAnalyzer and ErcChecker.

The PowerAnalyzer, which estimates power dissipation of electrical circuits, consists of 12,000 lines of C++. The PowerAnalyzer is composed of 3 related executables (PowerSrc, PowerCap, and PowerEst), and an application-specific library (`libPower`) of common functions and classes.

The ErcChecker performs electrical circuit checks such as checking for proper transistor ratios and fan-out limits. The ErcChecker tool consists of approximately 80,000 lines C++ code.

2.1. Cross-cutting Concerns

We are refactoring cross-cutting concerns as aspects. In order to provide necessary context for the refactoring-related faults, we describe several of the aspects below.

2.1.1. An Extra-Functional Timing Aspect

Extra-functional concerns contain functionality that is not part of the central task of an application [11]. The PowerAnalyzer contains a class, `Timer`, which is used throughout the code to record the time taken by processing steps. We re-implemented the calls using `around` advice to call the `Timer` class.

2.1.2. An ErcChecker Policy for the Query Hierarchy

The ErcChecker implements 58 different electrical checks as subclasses of an abstract base class `ErcQuery`. A static method in each class, `createQuerys()`, calls a common set of virtual methods in a manner similar to the Template Method design pattern [6].

For each query, `createQuery()` performs the same six conceptual steps. The first three steps identify circuit data, create instances of query objects, and call the `executeQuery()` method. The last three steps add failing queries to a container class (`LevelManager`), write query data to a log file, and delete queries that did not find electrical problems. Because of algorithmic and circuit-specific differences, each query has its own `executeQuery()` implementation.

We created a `QueryPolicy` aspect for steps 4 through 6 as after advice for the call to `executeQuery()`.

2.1.3. A Caching Aspect

Caching can improve performance of complex functions. An aspect-oriented solution is more modular and pluggable, and can be shared across many different cached functions [10].

We identified cached functions idiomatically since they use a static set or map. The ErcChecker contains 38 functions that separately implement caching. The aspect uses a list of pointcut expressions to specify where caching should be woven in. In addition, the aspect can enable or disable features such as cache profiling [13].

3. Unit and Integration Testing of Aspects

The process of aspectualization can be summarized with the following steps:

1. Inspect application code for duplicate, tangled code.
2. Create an aspect by using a small mock system to iteratively develop and unit test a prototype aspect.
3. Refactor the application to use aspects by removing duplicate or cross-cutting code from the application and then weaving the aspects.
4. Conduct integration testing of the refactored application by running regression tests.

The process is iterative — we repeat Steps 2, 3, and 4 to develop each aspect. At the end of the overall process many aspects can be created. This section focuses on the unit testing of aspects with the mock system and the integration testing of the aspectualized application with regression tests.

3.1. Coverage Goals

Our coverage goals depend on the phase of the aspectualization process. Testing the mock system with aspects (Step 2, above) aims to identify problems with the introduced aspects and how they are woven with the mock system. Integration testing of the refactored application (Step 4, above) focuses on regression testing.

Coverage of the Woven Mock System. To test the refactorings with the mock system, we want to ensure that each introduced aspect is covered by the test cases. We define *joinpoint coverage* to be that each advice body is tested when in context with each matching joinpoint. We also seek to satisfy statement coverage of the woven mock system.

Coverage of the Refactored Application. We evaluate joinpoint coverage for testing the mock system to verify that advice has been tested in all execution contexts. We relax the requirement of full system statement coverage. Since regression tests already exist for the original application, coverage of the core concern primarily measures the existing regression tests.

3.2. Unit Testing with a Mock System

The mock system is created *before* an aspect is implemented. The mock system's joinpoints must use naming conventions and structures that are consistent with the real application structure but on a much smaller scale: hundreds of lines of code instead of KLOC. Such similarity allows aspects to be moved with little or no change to the real application for weaving.

We can create the mock by implementing a small subset of the classes and methods of the real application. The mock system implements just enough functionality to test the aspects. The aspect is created and tested within the mock system. To perform unit testing, we weave the aspect with the mock system, and test until we satisfy joinpoint coverage.

Analysis of the weave identifies unused advice as an error. In addition, for each advice of each aspect, we annotate some methods or functions in the mock system to indicate whether or not they should have advice. These *advised methods* are checked to ensure that all advised methods have advice. Weave analysis is done during unit and integration testing.

3.3. Integration Testing of the Aspectualized Application

Once the aspects have been tested with the mock system, they are ready to be woven with the real application. The system is prepared for weaving by removing duplicated scattered code, and restructuring or renaming core concerns so that pointcut statements in the aspect can match the desired joinpoints in the program. The aspects are then woven with the application. We use the existing regression test suite associated with the application to perform integration testing with the aspects.

Joinpoints that are not executed by the integration tests are flagged as untested. If tests fail during integration testing, the aspect developer typically needs to determine the root cause of a defect. Suspected root causes can be simulated in the mock system before taking the time to modify, weave, and compile the real application.

The aspects are instrumented to gather joinpoint coverage data during integration testing, and the weave is checked for advised methods without advice and for unused aspect advice.

3.4. Tool Suite

We implemented three new tools to be used in our approach:

1. Advice Instrumentation: supports coverage analysis.
2. Aspect and System Coverage Measurement: analyzes dynamic test results to measure joinpoint coverage to check statement coverage of the mock system.
3. Weave Analysis: checks for unused advice or advised methods without advice.

The testing concepts are general and can be applied to other languages such as AspectJ. However, our tools are specific to AspectC++. They leverage features of the AspectC++ weaver, which writes information about the weave to an XML file for use by IDEs such as Eclipse¹. The XML weave file documents the methods and functions associated with advice and joinpoints [15].

Weave Analysis The weave analyzer parses the XML weave file to identify any unused aspects and unadvised methods. The XML file lists each aspect with its pointcuts and advice locations (with file identifiers and line numbers). Each joinpoint in the C++ program that matched to a pointcut is listed, along with a source code identifier and line number. This information is used to generate a list of unused advice. Weave analysis also checks annotations, which can specify that a method should or should not be advised. Annotations can also specify a particular aspect name rather than any aspect.

The list of annotated methods and functions is cross-referenced against the line numbers of function and method bodies associated with joinpoints in the weaver's XML weave file.

Advice Instrumentation Advice instrumentation provides us with coverage of aspects with respect to the context (joinpoint) in which they were executed. Existing statement coverage tools work on the woven code, but matching coverage to advice requires knowledge of the original (pre-weave) aspect and core concern code.

We preprocess the advice and insert a C++ macro with the original source line number and an AspectC++ construct, `JoinPoint::JPID`, which corresponds to the joinpoint id. This information allows us to compute joinpoint coverage, as explained in the next section.

¹www.eclipse.org

Coverage Analysis Joinpoint coverage data are generated during regression runs from code inserted during advice instrumentation. The generated data includes the original source code line number of the advice and the joinpoint `id`, which is cross-referenced with the weaver XML file to determine if joinpoints with advice were missed during testing.

Existing statement coverage tools can check coverage of all mock code during unit testing. We use `gcov`² on the woven code to measure statement coverage and filter out AspectC++-specific constructs from the results so that we can measure coverage of the original (pre-woven) system. The mock system coverage has helped us identify unnecessary mock system code (e.g., base class method that is always overridden) and untested code (e.g., error-handling code in the mock system).

4. Faults Uncovered During Unit Testing of Aspects

The following discussion describes faults that we encountered during aspect development in the mock system and how we found them.

4.1. Pointcut Too Strong

If the pattern used in a pointcut is too strong or restrictive, some required joinpoints may be missed [1]. The most extreme case of a pointcut being too strong is when it matches no part of the mock system. This occurred several times during initial development of an aspect (due to an incorrect pattern, or failure to account for C++ name-spaces, or an incorrect return type). The weave analysis identifies such unused advice as an error.

4.2. Pointcut Too Weak

A pointcut that is too weak results in advice matching too many joinpoints in the mock system [1]. Annotations can catch some of these joinpoints. For example, annotations used in the mock system that tested caching performance flagged advice that applied to too many functions.

Weak pointcuts are difficult to detect with annotations since finding them depends on how many annotations are in the mock system. Annotating every method could strengthen the analysis, but the cost of so many annotations, even in the mock system, would be very high.

Joinpoint coverage ensures that we test all contexts where advice is used. Advice woven in unexpected places can cause program state changes or output changes, which would then be caught by tests.

4.3. Incorrect Advice Behavior

Even when pointcuts are correct, the advice woven in can contain errors. Alexander et al. [1] give several examples of how this can happen, including failures related to postconditions, control flow, and program state. Joinpoint coverage helps detect these behavioral (run-time) errors by requiring that advice be tested in the context of each joinpoint of the mock system.

4.4. Advice Syntax Errors

Because AspectC++ uses a source-to-source weaver, the woven code must be compiled. While many syntax errors in advice cause a weave-time error, some errors in the advice body are not caught until compiling the woven code. As long as the advice is woven somewhere in the system, this error will be caught. Our approach helps this in two ways. First, unused advice is flagged. Second, using a small mock system provides fast feedback since mock systems compile in seconds, while compilation of the `ErcChecker` takes 15 minutes on our current system. We found this particularly helpful when using template-based advice and using the static joinpoint types available in AspectC++.

5. Potential and Observed Integration Faults

Faults can and did occur during the process of integrating the aspects into the application code. The faults types described in this section include actual faults that we discovered during integration testing, and faults that can potentially occur.

5.1. Inconsistent Renaming

Explicit calls to the `Timer` module were removed from the `PowerAnalyzer`. Because the code lacks common structure and naming conventions, the functions that used the `Timer` class were renamed from to begin with a common prefix (`tmr`) so that a single pointcut in the `TimeEvent` could match them.

We renamed the functions manually, although an Integrated Development Environment (IDE) could help automate this process. Failure to update calls to renamed methods will result in compiler errors, making this fault easy to find. Annotations were used to check the weave output to ensure that advice was woven to methods after renaming.

5.2. Missed Invocation of Extracted Methods

Another challenge was that some of the application code was written as large, procedural functions with many calls

²gcc.gnu.org/onlinedocs/gcc/Gcov.html

to the `Timer` module. For this function, Fowler’s “Extract Method” refactoring was first used [5], with the extracted method names beginning with `tmr` so that the pointcut would match them. When a new method is extracted, we must ensure that it is invoked where it is extracted. Statement code coverage can help identify dead code.

Using name-based pointcuts results in tight coupling that can break during maintenance due to name changes in functions [17]. This naming convention must be maintained over time so that exactly the desired functions are associated with the `TimeEvent` aspect. Since the `PowerEstimator`’s regression tests focus on functionality, an error associated with timing might not be immediately detected.

5.3. Common Policy Does Not Fit All Cases

When refactoring some `ErcQuery` subclasses in the `ErcChecker` to use an aspect, we found that some classes had a much simpler structure. This simplified structure meant that a method that the aspect always calls, `errorGenerated()`, was not used by those subclasses.

The aspect-based refactoring could insert a fault if the `executeQuery()` method of a subclass fails to set an object state variable that is used by the `errorGenerated()` method. We inspected the methods of all the sub-classes to validate that their respective `executeQuery()` methods set the required attribute used by the inherited method. Refactoring to an aspect did not introduce a defect, but it was a source for a potential defect.

5.4. Consequences from Base Class Changes

Some sub-classes of `ErcQuery` contained class-specific code that calls methods not inherited from the base class to record query-specific details used for validating and debugging the electrical checks.

We refactored the class-specific code into a new method, `logQueryDetails()` that is directly called by the policy aspect. This required modifying the base class to provide an empty default implementation for queries that do not need this functionality. Classes that need the functionality simply override the `logQueryDetails()` method.

Since adding an empty method to the base class is a one line change, we added the method directly to the C++ header file rather than using an `AspectC++` introduction. Creating `logQueryDetails()` required moving the query-specific code into the method body and changing method invocations to intra-class calls.

We must ensure that we removed all the query-specific logging from `createQueries()`, since any missed code would reference a deleted object and result in program termination. Integration testing must ensure that all modified queries are tested.

5.5. Accidental Code Duplication

The `QueryPolicy` aspect deletes query objects that do not find electrical errors. When refactoring core concerns, code that deletes these objects must be removed, or else a defect is introduced. Manual aspect-oriented refactoring is asymmetric, since the duplicated code (such as `delete`) must be manually removed, but the advice code is automatically woven in. One occurrence of this defect was found during integration testing, resulting in a memory fault. We modified the mock system to recreate this fault. Embedding a fault in the mock system uses the mock as a canonical example of incorrect usage and helps validate the root cause.

5.6. Broken Regression Tests

Refactoring scattered code to a single aspect standardized output logging. This caused system regression tests that use output logs to fail. Although output standardization should improve the maintainability of the `ErcChecker`, it does require a one-time update of the expected test output files and manual inspection of the changed output to validate its correctness.

5.7. Aspect Standardization Reveals Prior Faults

The `QueryPolicy` aspect is woven into all calls to `executeQuery()`, consistently applying the policy through advice. During refactoring, we found that one query failed to provide any information to the log file, but was otherwise correct. Changing to the aspect fixed this oversight, eliminating an existing subtle defect. Although this defect could have been found through inspection, using an aspect ensures the policy will be enforced.

6. Related Work

Aspect-oriented refactoring is being explored by many others [16, 7], including industrial systems [4]. Our work focuses on mock systems and on applying aspects to framework-based applications. Bruntink et al. have explored identifying aspects using clone detection tools [3].

We use two key ideas from test driven development: mock systems and using tests to drive development. To emulate a complex system dependency, test driven development may use a mock object (or *mock*) in place of the real object [2].

Lesiecki [9] uses mock objects and mock targets to help unit test aspects and uses visual markup in the Eclipse IDE to verify that pointcuts affected the expected program points. A mock target is similar to our concept of a mock system. However, a mock target is created from an aspect to

unit test pointcut matching. By contrast, our mock systems are created from the real system based on how we expect aspects to be used in that system.

Alexander, Bieman, and Andrews [1] describe key problems related to testing aspects and potential faults. Our joinpoint coverage is similar that used by Mortensen and Alexander [12]. Other proposed aspect coverage criteria include dataflow coverage [19], path coverage [8], and state-based coverage [18]. Dataflow and path coverage require static program analysis that is it beyond the scope of our work. Our legacy systems do not have state diagrams that could guide state-based testing.

7. Conclusions and Future Work

We presented a systematic approach to aspectualizing and testing large legacy systems. Mock systems enable aspect developers to quickly experiment with different pointcuts and advice, and iteratively develop and test aspects. Weave analysis and coverage-based testing help validate aspects within the mock system. Regression testing, weave analysis, and code coverage analysis ensure that aspects do not introduce new faults in the original system.

We are developing guidelines and patterns for creating and using mock systems when refactoring large systems. We are also interested in the evolution of refactored systems, and the co-evolution of mock systems. Pragmatic work can also explore the newly updated AspectC++ Eclipse plug-in to evaluate how well IDE features could be used along with our approach.

Our approach has focused on *before*, *after*, and *around* advice. We could also explore other using mock systems with other aspect constructs, including introductions, exception softening, and advice precedence.

References

- [1] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. *Technical Report CS-4-105, Dept. of Computer Science, Colorado State University*, March 2004.
- [2] D. Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003.
- [3] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, 2005.
- [4] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 50–59. ACM Press, Mar. 2003.
- [5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.
- [6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1995.
- [7] R. Laddad. Aspect-oriented refactoring part 1: Overview and process. Technical report, TheServerSide.com, 2003.
- [8] O. A. L. Lemos, J. C. Maldonado, and P. C. Masiero. Structural unit testing of AspectJ programs. In *2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)*, March 2005.
- [9] N. Lesiecki. Unit test your aspects. Technical report, Java Technology Zone for IBM's Developer Works, Nov. 2005.
- [10] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*. ACM, 2004.
- [11] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In D. H. Lorenz and Y. Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, Mar. 2005.
- [12] M. Mortensen and R. T. Alexander. An Approach for Adequate Testing of AspectJ Programs. In *2005 Workshop on Testing Aspect-Oriented Programs (held in conjunction with AOSD 2005)*, March 2005.
- [13] M. Mortensen and S. Ghosh. Creating pluggable and reusable non-functional aspects in AspectC++. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Bonn, Germany, Mar. 20 2006.
- [14] M. Mortensen and S. Ghosh. Using aspects with object-oriented frameworks. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006) Industry Track*, March 2006.
- [15] Olaf Spinczyk and pure-systems GmbH. *Documentation: AspectC++ Compiler Manual*, May 2005. <http://www.aspectc.org/fileadmin/documentation/ac-compilerman.pdf>.
- [16] P. Tonella and M. Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Trans. Softw. Eng.*, 31(10):819–832, 2005.
- [17] T. Tourwé, J. Bricchau, and K. Gybels. On the existence of the AOSD-evolution paradox. In L. Bergmans, J. Bricchau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.
- [18] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, March 2006.
- [19] J. Zhao. Unit testing for aspect-oriented programs. Technical Report SE-141-6, Information Processing Society of Japan (IPSJ), May 2003.