Unit-Testing Aspectual Behavior *

[Position Paper]

Cristina Videira Lopes and Trung Chi Ngo
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697
{lopes,trungcn}@ics.uci.edu

ABSTRACT

In the Java Aspect Markup Language (JAML), aspects are represented using two kinds of modules: a) regular java classes, encapsulating aspectual behavior; and b) XML binders, defining aspectual compositions. Besides leveraging the abundance of software engineering tools available for Java and XML, this approach for aspect-oriented programming also provides opportunities for testing aspects as independent units. In this position paper, JamlUnit, an extension of JUnit, is proposed as a framework for performing unit testing of aspects written in JAML. More specifically, this paper focus on testing aspectual behavior, i.e. behavior implemented in pieces of advice. JamlUnit uses mock objects that emulate execution context (join point) information. JamlUnit shows that performing unit testing on aspectual behavior is possible and relatively straightforward, as long as the AOP language observes a couple of simple requirements.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Testing

Keywords

Unit-Testing, Aspect-Oriented Language Design

1. INTRODUCTION

While AOP improves the modularity of certain crosscutting concerns, it introduces new problems with respect to verifiablity and testability. Testing aspect programs is hard. Most existing unit testing techniques are not applicable to testing aspects because of the tight coupling between aspectual behavior and its woven context. In addition, in aspect-oriented languages such as AspectJ [2], the idea of testing aspects as independent units in the traditional sense is not even appropriate, because aspects don't have independent identity or existence in those systems [1]. Therefore, most approaches to testing aspect-oriented programs have focused on integration testing, rather than on unit testing.

Our work aims at finding a solution for unit-testing aspects. We believe the challenges faced so far are a consequence of the aspect language designs, and are not inherent to the concept of crosscutting. Our approach is based on the Java Aspect Markup Language (JAML) [4], an extensible language framework we have developed for programming aspects. In JAML, aspectual behavior (e.g. tracing, security, etc.) is encapsulated in regular Java classes; the composition between base and aspectual behavior is defined in separate modules using an XML-based aspect language that captures pointcuts and advice definitions. Besides leveraging the abundance of software engineering tools available for Java and XML, and supporting aspect-specific language extensions [6], this approach has important consequences with respect to unit testing. In this paper we show how to unit-test aspectual behavior classes using JamlUnit, an extension of JUnit [5]. The next challenge, which is not addressed here, is to unit-test the XML binders, i.e. the pointcut definitions.

The rest of this paper is organized as follows. Section 2 discusses unit testing in general and the challenges of applying it to aspect-oriented programming. Section 3 presents the highlights of JAML and describes how to unit-test aspect behavior with JamlUnit. Section 4 discusses preliminary evaluation of JamlUnit. Section 5 presents related work, and section 6 concludes with future research directions.

2. UNIT TESTING

In this section, we discuss unit-testing and the obstacles faced in unit-testing aspects. Examples are presented throughout this section, as well as the rest of the paper, to illustrate the approach. Those examples are related to an inventory management program (IM), a GUI program that allows users to manage the inventory of a bookstore. Figure 1 shows the simplified UML class model of the program. The Inventory, Book, and Magazine classes model real objects in the store. The ListView class is a GUI form for displaying information about the inventory to end-users. The InventoryDatabase class provides operations for querying and updating data stored on a central database.

*This work has been partially supported by the National Science Foundation grant no.CCF-0347902.

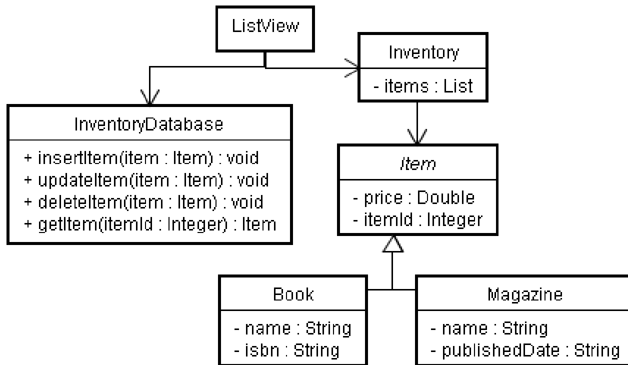

Figure 1: UML class model of the IM program

2.1 JUnit

Unit testing is a fundamental practice in most software development methodologies, including Extreme Programming [3]. In unit testing, programmers write test cases to exercise different parts of the system in isolation. Unit testing is usually performed before integration testing, to detect faults that pertain to the units, independent of the rest of the system.

JUnit [5] is a widely used framework for unit-testing Java programs. Figure 2 shows a sample JUnit test case for testing the *InventoryDatabase* class in Figure 1. With JUnit, a test case is usually implemented as a subclass of *junit.framework.TestCase*, which provides the basic functionality for unit testing. The implementation of a JUnit test case usually consists of a *setUp*(), a *tearDown*(), and a number of test methods. A test method must be public, return void, take no arguments, and have a name starting with the prefix "*test*". During the execution of a test case, JUnit uses Java's reflection API to find and invoke test methods. The *setUp*() and *tearDown*() methods are invoked before and after the execution of each test method, and are usually used to initiate and release test subjects (i.e. *test fixtures*).

Unit-testing non-trivial code that has some coupling with other parts of the system presents some challenges. *Mocking* [7] is a common solution for unit testing in such situations. This approach makes use of *mock objects*, or simply *mocks*, for replacing parts of the system with which the unit under test interacts. The implementation of such mock objects is usually simple and only meaningful to the test itself. For example, in order to unit test the robustness of a functional unit under database failures, mock objects can be created for emulating database errors instead of waiting for actual failures to occur.

2.2 Unit-Testing Aspects?

While aspect-oriented programming improves modularity by allowing the encapsulation of crosscutting concerns into units, testing such units poses significant challenges. First, let's define what unit testing means in the context of AOP. When using aspect modules, one would like to find answers to the following questions:

- Does an aspect implementation, i.e. behavior encapsulated in advice and introduction, do what it is supposed to do? This question is similar to the question we answer by unit-testing methods in OOP.
- Does a pointcut definition a) match anything?; b) include/exclude a given join point?; capture the join points, and only those, that it is supposed to capture? These questions are new to AOP.

```
1 // Simplified unit test case for testing the
2 // InventoryDatabase class. Details is obmitted for brevity
3 public class TestInventoryDatabase extends TestCase {
4
5  private InventoryDatabase inventoryDb;
6
7  protected void setUp() throws Exception {
8
9    // set up test fixtures
10   inventoryDb = new InventoryDatabase();
11 }
12
13 // test InventoryDatabase.getItem() method
14 // using conventional JUnit technique
15 public void test1() {
16   try {
17     // get the item with primary key = 1
18     // in the database
19     Item it = inventoryDb.getItem(new Integer(1));
20
21     // verify expected result
22     assertTrue(it != null);
23     assertEquals(it.getItemId().intValue(), 1);
24
25   catch (Exception ex) {
26     // the test case fail if exception occurs
27     fail();
28   }
29 }
30 }
```

Figure 2: Sample unit test case for InventoryDatabase

While recognizing that both questions are important, this paper focus only on the former. Testing aspect implementation is somewhat simpler than testing the pointcuts, but so far there have been no good solutions.

The obstacles for testing aspect implementation seem to come from design decisions that have been made in developing aspect-oriented languages, and not from any inherent property of aspects or crosscuts. AspectJ, the reference language of AOP, supports the concept of "aspect" as a non-instatiable unit that encapsulates both pointcut information and aspectual behavior. Without instantiation, unit testing, such as illustrated in Figure 2, becomes very difficult. Moreover, in AspectJ aspects are tighly coupled to the execution context information embodied in thisJointPoint object. This execution context information is provided by the underlying aspect execution framework and cannot be changed. Without being able to change it, one cannot *mock* it. Therefore, an aspect implementation cannot be isolated from all the classes that it applies to. Other languages have made similar decisions, making it very hard or impossible to unit-test aspects. But, as we will see, different design decisions enable unit-testing of aspects.

3. THE APPROACH

```
1 public class CachingInterceptor extends
2               edu.uci.jaml.lang.Interceptor {
3  private Hashtable cache = new Hashtable();
4
5  private Integer computeHash(Signature sig, Object[] args) {
6    // compute and return hash code of the given signature and
7    // args array. The implementatation is omitted for brevity
8  }
9
10 public Object aroundCachableFunc() {
11   Signature sig = thisJoinPoint.getSignature();
12   Object[] args = thisJoinPoint.getArgs();
13   Integer hashCode = computeHash(sig, args);
14   Object val = cache.get(hashCode);
15   if (val != null)
16      return val;
17   else {
18     val = thisJoinPoint.proceed();
19     cache.put(hashCode, val);
20     return val;
21   }
22 }
23 }
```

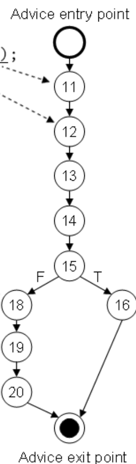

Figure 3: The implementation of a caching behavior

```
1  <aspect id="Caching">
2    <pointcut id = "p"
3       type="method-execution"
4       pattern = "Item InventoryDatabase.getItem(..)"/>
5
6    <interceptor class="CachingInterceptor">
7      <advice type = "around"
8        pointcut-refid="p"
9        interceptor-method = "Object aroundCachableFunc()"/>
10   </interceptor>
11 </aspect>
```

Figure 4: Binding instructions for a caching aspect

Our approach is based on JAML, an extensible language framework that we have developed. In this section, we briefly describe JAML, and then we introduce JamlUnit, the proposed approach towards unit-testing aspects written in JAML.

3.1 The Java Aspect Markup Language

A JAML program consists of three elements: core modules, aspectual modules, and aspect bindings [1].

- *Core modules*: Part of the system that implements the basic functionality of the system. These modules are developed in Java.
- *Aspectual modules*: Part that implements subsidiary concerns affecting a crosscut of the system. These modules are developed in Java, and are related to modules provided by JAML (by inheritance).
- *Aspect bindings*: XML-based binding specifications, written in JAML. They provide binding instructions that determine how core and aspectual modules are unambigiously composed to produce the the final behavior.

[1]JAML also supports the development of aspect-specific languages using standard plugin techniques [6], but that feature falls out of the scope of this paper.

Figures 3 and 4 show an example of a caching aspect written in JAML. In this example, the interceptor (Figure 3) contains a simplified implementation of caching behavior. The binder (Figure 4) determines points in the execution of the program specified by p need to be replaced by the around advice whose implementation is *CachingInterceptor.aroundCachableFunc*(). The implementation of the aroundCachableFunc() advice performs cache look up for already computed return value corresponding to the current join point's signature and list of arguments. The cached value is returned if one is found. Otherwise, the advised function will be proceeded normally, and its return value is cached for later usage.

JAML's join point model is derived directly from the join point model of AspectJ version 1.1 [2]. The current version of JAML supports all 16 types of primitive join points in AspectJ, except "*if*" and "*adviceexecution*". For example, the point cut p defined in Figures 4 depicts an AspectJ point cut "*execution(Item InventoryDatabase.getItem(..))*"

An interceptor definition provides a means of implementing AspectJ-like advice. It determines which method of the interceptor component needs to be invoked when certain join points are reached. JAML supports all five types of advice of AspectJ. In JAML, advice definitions are specified using <advice> elements, nested in an <interceptor> element. An interceptor component is implemented as a regular Java class that extends from *edu.uci.jaml.lang.Interceptor* class provided by the framework. Inherited from its super class, any interceptor has access to a protected field called *thisJoinPoint*. This special field provides reflective access to the context published by the currently executing join point.

An introduction definition allows users to insert new abstractions to existing Java classes throuch mix-in mechanisms. The current implementation of JAML supports method and constructor introductions, but not field introductions.

3.2 JamlUnit - Unit Testing Framework for JAML Programs

JamlUnit, an extension of JUnit, is proposed as a practical approach towards testing JAML programs. While the complete framework is aimed to support unit testing for both aspectual modules (e.g. interceptors and introductions) and aspect bindings (e.g. pointcut definitions), in this position paper, we focus on exploring ways to unit-test JAML interceptors. Unit-testing other elements is currently under development.

JamlUnit consists of a set Java helper classes that enable programmers to write regular JUnit test cases for testing aspect code as independent units. The underlying mecha-

```
 1 public class TestCaching extends TestCase {
 2  private CachingInterceptor t;
 3  private JoinPoint jp;
 4  private boolean proceedExecuted;
 5  private Object[] args;
 6
 7  protected void setUp() throws Exception {
 8
 9    // initialize test subject
10    t = new CachingInterceptor();
11  }
12
13  private void setUpThisJoinPoint() {
14    proceedExecuted = false;
15
16    // target, this, method name are
17    // not important
18    jp = new MockMethodExecJoinPoint(
19               null,    // target is ignored
20               null,    // this is ignored
21               args) { // args
22
23             public Signature getSignature() {
24
25               // call MockMethodExecJoinPoint.buildSignature()
26               // method to build a MockMethodSignature object
27               // that presents "Inventory.getItem(int)" method
28               return buildSignature(
29                 InventoryDatabase.class,    // target class
30                 "getItem",                  // method name
31                 new Class[] {Integer.class} // param types
32               );
33             }
34
35             public Object proceed() {
36               proceedExecuted = true;
37
38               // proceed always return 10
39               return new Integer(10);
40             }
41    };
42  }
43
44
45  // test CachingInterceptor as an independent unit
46  public void testaroundCachableFunc1() {
47    Integer result;
48    args = new Object[] {new Integer(5)};
49
50    setUpThisJoinPoint();
51    t.setThisJoinPoint(jp);
52    t.aroundCachableFunc();
53
54    // expect proceed to be run
55    assertTrue(proceedExecuted);
56
57    setUpThisJoinPoint();
58    t.setThisJoinPoint(jp);
59    result = t.aroundCachableFunc();
60
61    // expect proceed() to be skipped
62    // because cache value is found
63    assertFalse(proceedExecuted);
64
65    // expect the cached value to be 10
66    assertEquals(result.intValue(), 10);
67  }
68 }
```

Figure 5: Sample unit test case for CachingInterceptor.aroundCachableFunc() method

nism that makes this possible is that JamlUnit allows programmers to exercise aspectual behavior in carefully crafted environments as if certain actual join points in the target system are reached. Specifically, *mock join point objects* can be specified and bound to JAML interceptors for unit testing.

Figure 5 shows a sample JUnit test case for testing CachingInterceptor interceptor class. In this test case, the testaroundCachableFunc1() test method is written to test the aroundCachableFunc() method of the interceptor class. The implementation of this test method consists of two code fragments, lines 48-55 and 57-66, that exercise both branches of the control flow graph shown in Figure 3. Each code fragment does the following:

1. calls *setUpThisJoinPoint*() method to initialize the mock join point object. Lines 18-41 shows how a mock join point can be implemented. The *jp* join point object is instantiated with an anonymous class which extends the builtin *MockMethodExecJoinPoint* class and provides implementation for *getSignature*() and *proceed*() methods, as these methods are used during the execution of the *aroundCachableFunc*() method (see lines 11,12, and 18 of Figure 3).
2. calls the interceptor's *setThisJoinPoint*() method, which is defined in the *edu.uci.jaml.lang.Interceptor* class, to bind the mock join point to the interceptor.
3. invokes the interceptor's *aroundCachableFunc*() method to start executing the advice with the previously established mock join point.
4. uses JUnit's *assertXXX*() methods to verify the actual against expected results.

4. PRELIMINARY EVALUATION

4.1 Advantages

JamlUnit enables programmers to test aspectual modules of JAML programs as independent units. While faults in aspect systems can be detected by performing traditional unit and integration testing techniques against woven artifacts, JamlUnit shows the following advantages:

- **Cost-effective**: JamlUnit provides a cost-effective way to diagnose failures and detect faults that reside in aspectual implementations. While testing of woven artifacts can also help identifying the presence of those faults, localizing them can be dificult and time consuming. The number of test failures usually outnumbers that of the actual faults. For example, if a fault resides in an aspectual implementation that crosscuts ten different join points, test failures can be detected at ten different places in the system. Without appropriate supporting tools, aggregating such failures as they come from the same source is a challenging task. In addition, testing woven artifacts is the process of identifying faults during the aspect integration phase. Work in software testing suggests that fixing a bug in early phase is always better than fixing the same bug in

later phases. Thus, it makes sense to start identifying bugs at unit instead of integration level.

- **Aspect library development**: We conducted an informal survey of commonly referred aspects in both research (e.g. citation databases such as ACM Digital Library and IEEE Computer Society's Digital Library) and industry (e.g. print and online software development magazines, online discussion boards, and mailing list archives). Our findings show that the majority of those aspects are general purpose and orthogonal to core concerns. Example of such aspects are logging, tracing, persistence, profiling, and design patterns.

  From a developer's perspective, these aspects are perfect targets for developing general-purpose aspect libraries. In most aspect-oriented technologies, developping such libraries is possible; however, providing evidences of their correctness and reliability is a tricky task. There is no specific implementation of core concerns with which the aspects interact. A common practice for testing aspects in such circumstances is to weave and test aspects with different sample base programs. This approach has drawbacks. First, developping and maintaining such programs is usually non-trivial, tedious, and time-consuming. Second, a number of test programs, which is usually large, need to be developed to adequately test the aspects. Otherwise, those programs simply serve as "case studies" of the libraries, rather than sufficient proofs.

  JamlUnit facilitates the development of aspect libraries by providing effective testing support. Instead of developping a complete sample program, programers can develop a small fraction of its parts for emulating the execution context with which the libraries interact.

- **Light-weight**: JamlUnit is a light-weight framework that allows seemlessly integration with existing development methodologies for JAML. For programmers who know how to write unit test cases using JUnit, writing test cases with JamlUnit is relatively easy. In addition, existing software engineering tools available for JUnit can also be used with JamlUnit, including IDE-specific support for writing and running JUnit test cases.

4.2 Challenges

The major challenges of this approach are:

- **Selecting appropriate mock join points**. JamlUnit requires programmers to select and implement mock join points for unit-testing an aspect. Because the aspect may crosscut the base classes at many join points, the extensive testing approach in which all possible join points are investigated is impractical or even impossible. Thus, it is important to select meaningful mock join points that adequately represent the set of actual join points. This is a difficult task because such adequacy criteria has not been developed yet. Empirical studies and/or heuristics need to be devised.

- **Creating mock execution context**. This can be a lengthy and difficult task. The difficulty is inherent to the nature of aspects, especially for those that interact with core concerns. Future work in JamlUnit will tackle this problem by providing support tools for writing mock execution context. For example, such tools might perform dependency analysis between an aspectual behavior and its execution context to generate code skeletons for implementing mock execution context.

Moreover, there are limits for what unit testing can do. Testing aspects in isolation cannot detect faults resulting from the itegration of aspects with the implementation of core concerns, such as faults identified in [1]. This difficulty is inherent to unit testing in general. Unit testing should be followed by integration testing.

4.3 JamlUnit and AspectJ

AspectJ is the reference language for aspect-oriented programming, and deserves special attention. Different techniques for testing woven artifacts in AspectJ has been developed, such as state-based testing [8] and data-flow-based testing [9]; however, litle is known how to test AspectJ aspects as independent units. The question is "*can we develop JamUnit-like unit testing framework for AspectJ?*". Our investigation of AspectJ shows that the answer is "*no*" because of the following issues:

- **AspectJ's aspects do not have instantiable existence**. In AspectJ, aspects cannot be instantiated as independent units of execution. It is not possible to instantiate aspectual behavior for testing, such as what is shown in line 10 of Figure 5.

- **AspectJ provides limited write access to execution context information**. AspectJ's reflective API allows programmers to alter several types of contextual information published by the current join point for around advices. However, invasively replacing the execution context like shown in line 51 of Figure 5 is not possible in AspectJ.

4.4 General Requirements for unit-testing aspectual behavior

Our premilinary experience with JamlUnit shows the possibility of devising clean aspect testing methodology for JAML or other aspect-oriented technologies using mock object mechanisms, in which the implementation of crosscutting concerns (i.e. aspectual implementation) can be tested in isolation. Such methodologies and/or target AO technologies need to provide adequate support for the following issues:

- **Aspect implementations must be treated as first-class citizens**. In most existing aspect-oriented technologies, an aspect does not have independent identity or existence because of its tight coupling with other entities and/or execution context[1]. In order to test aspects as units, this coupling must be loosened. One potential solution for this problem is to take the route favored by JAML, in which aspectual behavior is separated from binding instructions, and are defined as regular component units.

- **Join points can be explicitly specified and bound to aspectual behavior**. The idea of using mock objects for unit and integration testing is not so novel. This technique has been studied and used for years. However, applying it to testing aspects remains challenges. Argubally, these challenges are because of accidental rather than essential difficuties. The design of most existing aspect languages does not provide adequate support for *stuffing* aspectual implementation with mock execution context. By providing reflective API for explicitly constructing join points and binding them to aspectual implementation, JAML shows initiate steps towards solutions for this issue.

5. RELATED WORK

In [1] Alexander *et al.* identify four sources of faults in an aspect program, which can be summaried as follow: 1) faults that reside in the implementation of core concerns and unaffected by aspectual implementation, 2) faults that reside in aspectual implementation, independent from the execution context, 3) faults are emergent properties created when one or more aspects are woven into the primary abstraction. Our work focuses on devising unit test technique for identifying faults in the second category. As faults in the first category can be detected with traditional testing techniques and our intention does not stop at unit testing level, our future work involves faults in the third category. The candidate fault model and associated testing criteria proposed in [1] is excellent for devising new techniques for testing woven artifacts.

[8, 9, 10] investigate techniques for testing aspect-oriented programs. While these publications focus of exploring ways to test woven artifacts, our work in JamlUnit is aimed at testing aspects in isolation.

6. CONCLUSIONS

JamlUnit is proposed as a practical approach for unit-testing aspectual behavior. JamlUnit provides a set of helper classes for writing regular JUnit test cases to unit-test aspect implementations. The underlying mechanism of JamlUnit is the use of mock execution context for isolating the aspectual behavior under test.

The main contribution of this work is that it shows the possibility and requirements for devising clean unit testing techniques for aspect-oriented programs using mock object mechanisms. Beside addressing challenges specified in section 4.2, future work in JamlUnit involves devising techniques for testing other elements of JAML, including aspectual bindings and introdutions.

7. REFERENCES

[1] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testingb of aspect-oriented programs. Technical report, Department of Computer Science, Colorado State University, 2004.

[2] AspectJ. http://www.eclipse.org/eclipse, 2004.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] JAML. http://www.ics.uci.edu/~trungcn/jaml/, 2004.

[5] JUnit. http://www.junit.org, 2004.

[6] C. V. Lopes and T. C. Ngo. Jaml: An extensible language framework for developing aspect-specific languages. submitted to ecoop'05, 2005.

[7] M. Objects. http://www.mockobjects.com, 2004.

[8] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspect-oriented programs. Technical report, Department of Computer Science, Colorado State University, September 2004.

[9] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, pages 188–197, November 2003.

[10] Y. Zhou, D. Richardson, and H. Ziv. Towards a practical approach to test aspect-oriented software. In *Net.ObjectiveDays 2004 Workshop on Testing Component-based Systems (TECOS 2004)*, Sep 2004.