A Novel Approach to Unit Testing: The Aspect-Oriented Way

Guoqing Xu and Zongyuan Yang
Software Engineering Lab, Department of Computer Science
East China Normal University
3663, North Zhongshan Rd., Shanghai 200062, P.R. China
{gqxu_02,yzyuan}@cs.ecnu.edu.cn

ABSTRACT

Unit testing is a methodology for testing small parts of an application independently of whatever application uses them. It is time consuming and tedious to write unit tests, and it is especially difficult to write unit tests that model the pattern of usage of the application they will be used in. Aspect-Oriented Programming (AOP) addresses the problem of separation of concerns in programs which is well suited to unit test problems. On the other hand, unit tests should be made from different concerns in the application instead of just from functional assertions of correctness or error. In this paper, we firstly present a new concept, application-specific Aspects, which mean top-level aspects picked up from generic low-level aspects in AOP for specific use. It can be viewed as the separation of concerns on applications of generic low-level aspects. Second, this paper describes an Aspect-Oriented Test Description Language (AOTDL) and techniques to build top-level aspects for testing on generic aspects. Third, we generate JUnit unit testing framework and test oracles from AspectJ programs by integrating our tool with AspectJ and JUnit. Finally, we use runtime exceptions thrown by testing aspects to decide whether methods work well.

KEYWORDS

Aspect-Oriented Programming, Unit Test, Aspect-Oriented Test Description Language

1. INTRODUCTION

There is a growing interest in applying program testing to the development process, as reflected by the Extreme Programming (XP) approach [1]. In XP, unit tests are viewed as an integral part of programming. Tests are created before, during, and after the code is written — often emphasized as "code a little, test a little, code a little, and test a little ..." [2]. The philosophy behind this is to use regression tests [3] as a practical means of supporting refactoring.

A unit test suite comprises a set of test cases. A test case consists of a test input and a test oracle, which is used to check the correctness of the test result. Developers usually need to manually generate the test cases based on written or, more often, unwritten requirements. Some commercial tools for Java unit testing, such as ParaSoft's Jtest [4], attempt to fill the gaps not covered by any manually generated unit tests. These tools can automatically generate a large number of unit test inputs to exercise the program. However, no test oracles are produced for these automatically generated test inputs unless developers do some additional work: in particular, they need to write some formal specifications, runtime assertions [5] or more practically, make program invariants generated dynamically with a certain tool like Daikon [6] and use these invariants to improve the test suites generation [7, 8]. However, with the current formal assertions, it is very difficult to generate tests that can model some non-functional features of the program, e.g. the performance of the program and temporal logic of methods' execution.

Aspect-Oriented Programming (AOP) addresses the problem of separation of concerns in programs [9, 10, 11, 12]. Since in AOP, the crosscutting properties are monitored to reflect the program from different aspects, a lot of tasks which have been difficult to be handled in traditional ways

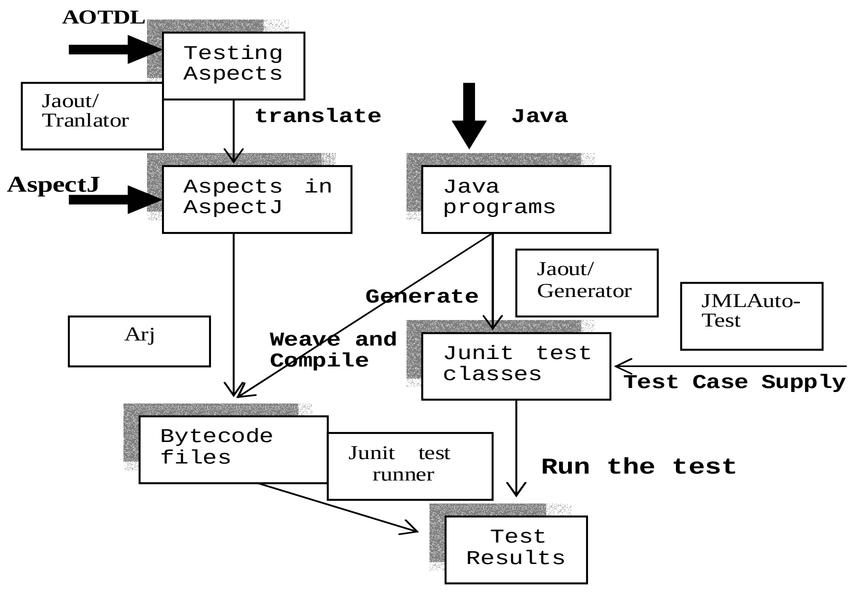

Fig.1. An Overview of the basic technique

are easily done. For example, the performance and methods' execution order problems have been well solved in AOP. Therefore, using a crosscutting property of the program as the criterion to check the correctness of the application in the corresponding aspect is well suited to the unit testing problems.

However, currently in AOP, programmers build generic aspects to monitor certain crosscutting properties for a wide variety of uses including program tracing, rum time assertion checking and etc. This makes it difficult for testers to identify them and make those for testing as test oracles. Therefore, how to build specific testing aspects which can be identified as test oracles becomes the key problem in making the aspect-oriented unit testing practical.

We attempt to solve this problem by collecting those aspects in AOP for the same use into application-related top-level aspects. We call these aspects the application-specific aspects. For example, aspects for tracing can be collected and made into Tracing Aspects, aspects for pre and post condition assertions are to be built as Assertion Aspects and etc. Building application-specific aspects can be viewed as the separation of concerns on applications of generic aspects. A kind of application-specific aspects share some common features. In this way, we can build Testing Aspects which will send runtime messages which can be received by unit test programs and identified as test oracles.

Fig.1 illustrates the basic technique used in our approach to generating the unit test. The testing aspects described by Aspect-Oriented Test Description Language (AOTDL) can be translated by our tool JAOUT/translator as low-level aspects in AspectJ. Then after weaved with Java programs, the aspects are compiled to bytecode files (class files). The tool JAOUT/generator generates the corresponding JUnit test classes from the program to be tested. These test codes can serve as test oracles. Finally, fed with the test inputs generated automatically by JMLAutoTest [14], the unit test is run and the results are judged by the runtime exceptions thrown from the testing aspects we made.

The rest of this paper is organized as follows: section 2 briefly introduces the basic concepts in AspectJ and section 3 describes how to build testing aspects with AOTDL. We present the generation of test oracle and test cases in detail in section4. After section 5 describes the related work, we conclude our approach and describe the future work in section 6.

2. AspectJ

In this paper, we use AspectJ as our target language to show

```
Class Stack{
  public void init(){...}
  public void push(Node n){...}
  ...
}

Aspect TempLogic{
  protected boolean isInitialized = false;
   //method push is called
  pointcut pushReached(Stack st):
       target(st)&&call(void
Stack.push(Node));
  //method init is called
  pointcut initReached(Stack st):
       target(st)&&call(void
Stack.init(void));
  //advice after init is called
  after(Stack st):initReached(st){
    isInitialized = true;
 }
  //advice before push is called
  before(Stack st)
        throws NotInitializedException:
  pushReached(st){
    if(!isInitialized)
     throw new NotInitializedException();
 }
}
```

Fig. 2. An Aspect providing advices for the temporal logic of methods execution in *Stack*

the basic idea of aspect-oriented unit testing. AspectJ [13] is a seamless aspect-oriented extension to Java. AspectJ adds some new concepts and associated constructs are called join points, pointcuts, advice, and aspect. The join point is an essential element in the design of any aspect-oriented programming language since join points are the common frame of reference that defines the structure of crosscutting concerns. The join points in AspectJ are well-defined points in the execution of a program. A pointcut is a set of joint points that optionally exposes some of the values in the execution of these joint points. AspectJ defines several primitive pointcut designators can be defined according to these combinations.

Advice is a method-like mechanism used to define certain codes that is executed when a point cut is reached. There are three types of advice, that is, *before*, *after* and *around*. In addition, there are two special cases of after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point. *Aspects* are modular units of crosscutting implementation. *Aspects* are defined by aspect declarations, which have a similar form of class declarations. Aspect declarations, as well as other declarations such as method declarations, that are permitted in class declarations.

An AspectJ program is composed of two parts: (1) non-aspect code which includes some classes, interfaces, and other language constructs as in Java, (2) aspect code which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that aspect and non-aspect code run together in a property coordination fashion. Such a progress is called aspect weaving and involves making sure that applicable advice runs at the appropriate join points. Fig.2 illustrates a sample aspect which provides advices for the methods execution order in the class *Stack*. The advices defined in this aspect require that the *push* method can not be executed if the *init* method is not called.

3. BUILDING TESTING ASPECTS

In this section, we present how to build the application-specific aspects for testing with AOTDL.
Prepare your submissions on a typesetter or word processor.

3.1 AOTDL

AOTDL explicitly specifies the advices for the criteria of meaningless test cases and test errors. Fig.3 illustrates the AOTDL representation of the TempLogic aspect which we have mentioned above. The major differences between two representations in Fig.2 and 3 lie with the fact that the aspect in Fig.2 is a low-level AspectJ aspect which can be of any uses, e.g. tracing, asserting, logging and whatever. But the aspect in Fig.3 is the application-specific aspect which is specifically built for testing. It can be viewed as the separation of concerns on applications of generic aspects. The *Meaningless* and *Error* units contain the advices for criteria of meaningless test cases and test errors respectively. We use the syntax as follows:
advicetype(arguments): *pointcuts*: *conditions*: *message*
conditions is a boolean expression which means the conditions in which the current test case is meaningless or the current test fails. The *message* means the printed message

```
TestingAspect TempLogic{
 // all pointcuts and other utility advices are
declared
 // in the Utility unit
  Utility{
   protected boolean isInitialized = false;
  //push is reached
   pointcut pushReached(Stack st):
     target(st)&&call(void
Stack.push(Integer));
  //init is reached
   pointcut initReached(Stack st):
     target(st)&&call(void Stack.init(void));
   after(Stack st):initReached(st){
     isInitialized = true;
   }
  }
  MeaninglessCase Advice{
   //advices for specifying criteria of
   //meaningless test cases
    before(Stack s):
        pushReached(s):
             s.getSize()>=MAX:"Overflow";
    ...
 }
  Error Advice{
  //advices for specifying criteria of test
  //errors
   before(Stack s):
        pushReached(s):
          !isInitialized:"Not Initialized";
    ...
 }
}
```

Fig.3. The AOTDL representation of the TempLogic Aspect

when the conditions happen. To enhance the expressiveness of *conditions* clause, AOTDL supports most kinds of boolean expressions used in formal languages predicts [15] including *forall*, *exist*, function calls returning boolean values and etc. And we are still working on supporting more expressions to make the language convenient and expressive enough for testers. The declarations of pointcuts, fields and all the other advices which do not directly affect the criteria of what is a meaningless case and what is a failed test are put into the *Utility* unit.

Although the syntax of AOTDL seems simple, it bridges the gap between abstract application-specific aspects of the program and detailed language level aspects in AOP, and hence can be viewed to help the separation of concerns on applications of generic aspects in AOP. This idea can also be used in making other application-related tools to extract top-level application aspects from the generic aspects.

```
public Aspect TempLogic{
  // Definitions in Utility unit are not changed
   protected boolean isInitialized = false;
   //push is reached
   pointcut pushReached(Stack st):
     target(st)&&call(void Stack.push(Integer));
     //init is reached
     pointcut initReached(Stack st):
       target(st)&&call(void Stack.init(void));
     after(Stack st):initReached(st){
     isInitialized = true;
   }
  //meaningless advices
  before(Stack s)
      throws MeaninglessTestInputException:

  pushReached(s){
   if(s.getSize()>=MAX){
    MeaninglessTestInputException ex= new
     MeaninglessTestInputException("overflow");
        ex.setSource("TempLogic");
    }
  }
  //error advices
  before(Stack s)throws TestErrorException:
                   pushReached(s){
    if(!isInitialized){
       TestErrorException ex =new
          TestErrorException("Not Initialized");
       ex.setSource("TempLogic");
  }
 }
}
```

Fig.4. The Translation of TempLogic Testing Aspect

3.2 Translation of Testing Aspects

JAOUT/translator is made to translate the testing aspects represented by AOTDL into generic aspects in AspectJ. The translation of TempLogic testing aspect shown in Fig.3 is illustrated in Fig.4. Definitions in *Utility* unit are not changed since they do not affect test oracles. The advices defined in *meaningless* and *error advice* unit throw a *MeaninglessTestCaseException* and *TestErrorException* respectively when the specified conditions are reached.

Now we make clear that AOTDL is the extension of AspectJ in making application-specific aspects for testing. Our translator also supports the mixture of the testing aspect specified by AOTDL and generic aspects in AspectJ. After translated by JAOUT/translator and then weaved with the *Stack* program and compiled by arj, the corresponding byte codes (.class) files are generated.

4. TEST ORACLES AND TEST CASES GENERATION

This section presents the details of our approach to automatically generating a JUnit test class from a Java class weaved with aspects. We firstly describe how test outcomes are determined. Then we describe the protocol and techniques for the user to supply test data to generated test oracles. Finally, we discuss the automatic generation of test methods and test classes.

4.1 Determining test outcomes

The outcome of a call to M for a given test case is determined by whether translated testing aspects throw exceptions during M's execution, and what kind of exception is thrown. If no exception is thrown, then the test case succeeds (assuming the call returns), because there was no meaningless case/error found, and hence the call must have satisfied specifications defined in testing aspects.

Similarly, if the call to M for a given test case throws an exception that is not an *MeaninglessTestInput* or *TestError* exception, then this also indicates that the call to M succeeded for this test case since they are thrown by other aspects or the method itself, but not testing aspects. Hence if the call to M throws such an exception instead of a meaningless or error exception, then the call must have satisfied specifications of testing aspects. With JUnit, such exceptions must, however, be caught by the test method testM, since any such exceptions are interpreted by the framework as signaling test failure. Hence, the testM method must catch and ignore all exceptions that are not meaningless and error exceptions.

If the call to M for a test case throws a *TestError* exception, then the method M is considered to fail that test case. If a *MeaninglessTestInput* exception is caught, the current test case is considered as a meaningless test case and therefore, is ignored.

4.2 Supplying Test Cases

Given a Java class *M.java*, JAOUT/Generator generates three classes: *M_Aspect_Test*, *M_Aspect_TestCase* and *M_Aspect_TestClient*. *M_Aspect_Test* is a JUnit test class to test all methods in class M. *M_Aspect_TestCase* is a test case provider, which extends *M_Aspect_Test* and initialize test

```
public void testM() {
   final A1[] a1 = vA1;
     : : :
   final An[] an = vAn;
   for (int i0 = 0; i0 < receivers.length;
                                    i0++)
     for (int i1 = 0; i1 < a1.length; i1++)
        : : :
        for (int in = 0; in < an.length; in++)
           {
          if (receivers[i0] != null) {
           try {
              receivers[i0].M(a1[i1],
                       ..., an[in]);
           }
           catch
           (MeaninglessTestInputException e)
           {
            /* ... tell framework the test case
              was meaningless ... */
             result.meaninglesscases++;
             continue;
           }
        /*  ... tell framework the current
             test fails*/
           catch (TestErrorException e) {
                String msg =
                   "In testing aspect  "+
                           e.getSource()+":"
                              +e.getMessage();

                 system.err.println(msg);
                 result.errors++;
           }
           catch (java.lang.Throwable e) {
                // success for this test case
                 continue;
           }
           finally {
              setUp(); // restore test cases
           }
       } else {
             /* ... tell framework test case
           was meaningless, since the test
           cases are not initialized ... */
      }
      ...
    }
  }
}
```

Fig.5. The test method for the corresponding method M

fixture. *M_Aspect_TestClient* is the test client in which test cases can be generated automatically by JMLAutoTest [14] tools.

4.2 Test Methods

There will be a separate test method, testM for each target instance method (non-static), M, to be tested. The purpose of testM is to determine the outcome of calling M with each test case and to give an informative message if the test execution fails for that test case. The method testM accomplishes this by invoking M with each test case and indicating test failure when the testing aspects throw a *TestError* exception. Test methods also note if test cases were rejected as meaningless when a *MeaninglessTestInput* exception was caught.

What Fig.5 illustrates is the test method for the method M. The differences between this method and the test method in Jmlunit [5] framework are that the *MeaninglessTestInputException* and *TestErrorInputException* are replaced for the *PreconditionEntryError* and *PostconditionError* and the exceptions used in our testing framework are thrown by **Testing Aspects** instead of a runtime assertion checker. Since patterns difficult to be modeled with traditional formal specifications are well treated as recognized as crosscutting properties in aspects, our approach may be adapted in more situations than a testing framework based on formal specifications exemplified by Jmlunit.

5. CONCLUSIONS

Automatically generating unit tests is an important approach to making unit test practical. However, existing specification-based tests generation methods can only test the program behavior specified by invariants. Since the program invariants only focus on functional behaviors, patterns related to non-functional properties of the program can not be modeled, and therefore, existing specification-based test generation can not test these special aspects of the program.

In the paper, we take a different perspective and present an approach to generating the unit testing framework and test oracles from aspects in AOP. First, we describe a new concept, application-specific aspect, which means the top-level aspect picked up from generic aspects in AOP. This can be viewed as the separation of concerns on specific application of common AOP's aspects. Then we discuss an Aspect-Oriented Test Description Language (AOTDL) to build the application-specific aspects for testing, namely testing aspects. AOTDL explicitly specifies the properties for testing which can be translated into the common aspects in AspectJ. After weaving and compiling programs, we automatically generate the unit testing codes which can serve as test oracles. Finally, test outcomes are decided on different exceptions thrown by testing aspects. Since we integrate our tool with JMLAutoTest, testers can make the test cases generated automatically and use the double-phase testing way to filter out the meaningless test cases.

REFERENCES

1. Beck K., Extreme Programming Explained. Addison-Wesley, 2000.
2. Beck K. and Gamma E., Test infected: Programmers love writing tests. *Java Report, 3(7),* July 1998.
3. Korel B. and Al-Yami A. M., Automated regression test generation. *In Proc of ISSTA 98,* Clearwater Beach, FL, pages 143–152. IEEE Computer Society, 1998.
4. Parasoft Corporation, Jtest manuals version 4.5, http://www.parasoft.com/, October 23, 2002.
5. Cheon Y. and Leavens G. T., A simple and practical approach to unit testing: The JML and JUnit way, *In Proc of 16th European Conference Object-Oriented Programming (ECOOP02), pp. 231-255.*, 2002.
6. Ernst, M. D., Cockrell, J., Griswold, W. G., Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering, vol. 27, no. 2,* Feb. (2001) 1-25
7. Xie T. and Notkin D., Tool-Assisted Unit Test Selection Based on Operational Violations, *In Proc of 18th IEEE International Conference on Automated Software Engineering (ASE'03),* Montreal, Canada, Oct. 2003.
8. Xie T. and Notkin D., Mutually Enhancing Test Generation and Specification Inference. *In Proc of 3th International Workshop on Formal Approaches to Testing of Software (FATES'03),* Montreal, Canada, Oct. 2003.
9. Bergmans L. and Aksit M, composing crosscutting concerns using composition filters, *Communications of the ACM, Vol.44, No.10, pp.51-57,* Oct.2001.
10. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., and Irwin J.. Aspect-oriented programming. *In Proc. ECOOP'97, LNCS vol. 1241,* Springer-Verlag, 1997.
11. Lieberherr K.J., Orleans D., and Ovlinger J., Aspect-Oriented Programming with Adaptive Methods, *Communication of the ACM, Vol.44, No.10, pp.39-41,* Oct.2001.
12. Ossher H. and Tarr P., Multi-Dimensional Separation of Concerns and the Hyperspace Approach, *In Proc. the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development,* Kluwer, 2001.
13. The AspectJ team, The AspectJ Programming Guide, http://eclipse.org/aspectj, May 2004.
14. Xu G. and Yang Z., JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit., *In Proc. 3rd International Workshop on Formal Approaches to Testing of Software(FATES'03), pp. 118-127,* Montreal, Canada, Oct.2003., also in *LNCS vol.2931*, Springer-Verlag, Jan. 2004.
15. Leavens G. T., Baker A. L., and Ruby C.. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001.)