FlexTest: An Aspect-Oriented Framework for Unit Testing

Dehla Sokenou and Matthias Vösgen

Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik,
Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Softwaretechnik,
Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin,
EMail: {dsokenou|mvoesgen}@cs.tu-berlin.de

Abstract. This paper examines whether test problems that occur specifically during unit testing of object-oriented programs can be solved using the aspect-oriented programming paradigm.
It presents the various problems in unit testing, shows conventional solutions and describes aspect-oriented solutions to the problems. The aspect-oriented solutions are supported by the unit test framework *FlexTest* of which the paper gives an overview.

1 Introduction

Unit testing is popular in the domain of object-oriented software development. Unit tests are written using the same implementation language as is used for the application under test.

However, some problems occur when unit testing object-oriented programs. There are two different reasons for this. Firstly, problems result from object-oriented characteristics like encapsulation and inheritance. Secondly, problems are caused by unit testing itself. Writing test cases directly in an implementation language is easy for programmers but it is sometimes a repetitive task. All constraints of the used language hold for the test, too.

In this paper, we investigate aspect-oriented programming techniques as a solution to a few of the problems encountered in unit testing. We present the various problems in unit testing, show conventional solutions and describe aspect-oriented solutions to the problems.

The unit test framework *FlexTest* [1] is presented, which supports the given solutions. For each solution, we give an example of how it is implemented in *FlexTest*.

The paper is organized as follows. Section 2 gives a brief introduction to aspect-oriented programming. Section 3 looks at the question of wether unit testing is a cross-cutting concern in the sense of aspect-oriented programming with regard to the application under test. In Section 4, we consider the unit test framework *FlexTest* in relation to the given problems and their solutions. Section 5 compares our work with similar approaches. Finally, we give a conclusion and suggest some entry points for discussion in Section 6. This section also includes an outlook on future work.

2 Aspect-Oriented Programming Techniques

Some concerns cannot be encapsulated in a class using object-oriented software development. Classical examples here are logging, security, and synchronization. For example, the logging server is implemented in one class, but client code is needed in all classes that support logging. We say that code is scattered over the system and is tightly coupled or tangled with the system. If both characteristics, scattering and tangling, apply to a concern we call it a cross-cutting concern.

Aspect-oriented programming is an extension of object-oriented programming that provides a solution for the given problem. Cross-cutting concerns are encapsulated in modules called aspects. Scattering and tangling are hidden in the source code. An aspect weaver is used to integrate cross-cutting concerns into the business logic of the system. Source code is not changed but the compiled or run-time code is modified, depending on the weaving strategy. Aspect-oriented programming provides a non-invasive way of adding new functionality to an implementation without affecting the source code.

An aspect encapsulates method-like code fragments, called advices. To weave new functionality into an existing (adapted) system, join points have to be defined. A join point is a point in the control flow of the adapted system, e.g. a method call, method execution or an access to an instance variable. Important for this paper is also the `cflowbelow` statement that refers to all points in the stack trace below a given control-flow point. Point-cuts are collections of join points. Advice code can be executed before, after or instead of the code addressed by a point-cut, e.g. instead of a referred method execution with the `around` statement. The original method can be called within the `around` advice using the `proceed` statement.

We use aspect-oriented programming techniques for testing object-oriented systems. We regard testing as a cross-cutting concern. Aspect-oriented programming helps to encapsulate test code and provides a flexible test instrumentation solution.

Before presenting some applications of aspect-oriented programming techniques in our unit testing framework *FlexTest* in Section 4, we give a brief introduction to aspect-oriented unit testing in the following section.

3 Aspect-Oriented Unit Testing

This paper investigates the use of aspect-oriented programming techniques for unit testing. We view unit testing as it is defined in the field of extreme programming. The focus of a unit test is a class, "but test messages must be sent to a method, so we can speak of method scope testing" [2]. For each method, the tester must implement independent unit tests. A unit testing framework is generally used to automatically execute and evaluate test cases.

Using aspect-oriented techniques for unit testing assumes that testing is a cross-cutting concern with respect to the implementation under test (IUT). In most cases, testing involves inserting additional test code into the IUT.

By looking at the additional test code and the IUT, we are able to determine that

- We must abstract from the context of the methods under test in the testing phase.
- Methods are dependent on instance variables. In some cases, test code must have privileged access to instance variables to initialize and evaluate test cases.
- Test code must be inserted in certain methods and classes in a similar way.
- Normally, unit test code is removed after the testing phase.

Given these requirements, we can say that testing is a cross-cutting concern with regard to the system under test. Both characteristics of aspects, scattering and tangling, hold for test code.

If, then, testing is a cross-cutting concern, the use of aspect-oriented programming techniques will yield benefits.

The next section shows some applications of aspect-oriented programming in unit testing and presents our unit test framework *FlexTest*.

4 Aspect-Oriented Programming Techniques in *FlexTest*

In this section, we examine some selected problems encountered while unit testing object-oriented systems. First, conventional solutions to a presented problem are shown and then these are compared with the aspect-oriented solution provided by *FlexTest*.

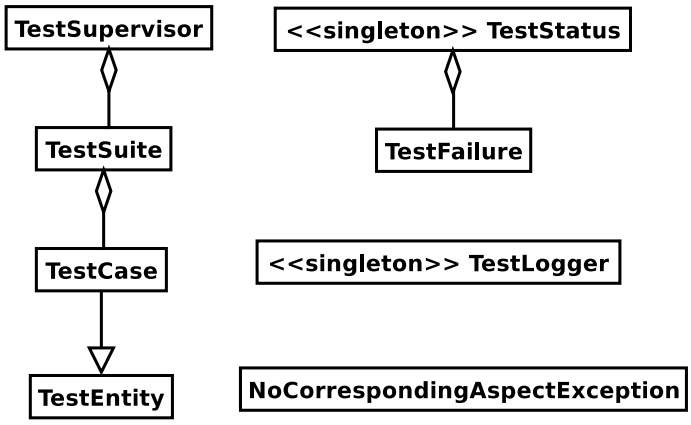

Fig. 1. Class Hierarchy of Flextest

FlexTest is a unit test framework for testing object-oriented programs. It has two parts, an object-oriented and an aspect-oriented part. The object-oriented

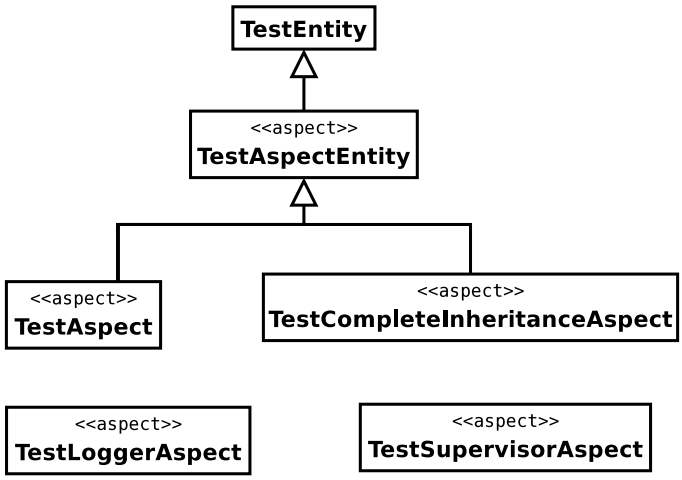

Fig. 2. Aspect Hierarchy of Flextest

part is similar to the JUnit test framework [3] and provides classes for defining test cases and test suites and functionality for test execution (see Fig. 1). Examples of framework classes are `TestCase` and `TestSuite`, similar to JUnit. The aspect-oriented part supports additional functionality for using aspect-oriented techniques in *FlexTest* (see Fig. 2). The main class is the class `TestAspect`. When using the `FlexTest` framework, the class `TestCase` and the aspect `TestAspect` have to be subclassed. Each subclass of `TestAspect` is assigned to a subclass of `TestCase`. When we refer to classes of the implementation under test in the following sections, we annotate them with *(IUT)*, the ones of the object-oriented part of the framework with *(OOF)* and the aspects of the aspect-oriented part with *(AOF)*. Subclasses of framework classes and aspects are annotated like their superentities.

We decided to develop a new framework instead of using an existing framework like JUnit because this allows us to design the object-oriented part depending on the aspect-oriented part and to enhance the aspect-oriented part flexibly with respect to the unit test problems.

In the following subsections, we look at the different problems encountered in unit testing. The aspect-oriented part of the *FlexTest* framework provides aspect-oriented solutions to the given problems, as described below. The IUT is a library of linear-algebra algorithms written in Java. The examples given in the following subsections are written in AspectJ [4].

4.1 Encapsulation

In object-oriented systems, attributes and methods are encapsulated in classes. As a result of encapsulation, classes are only accessible by interfaces. In testing,

encapsulation leads to the problem of insufficient access to the IUT. Classes can only be treated as black boxes without access to inner states, which leads to a less precise test oracle. Private methods cannot be tested by the test framework, which is a test driver problem.

Without using external tools, there are two main solutions to this problem:

1. **Changing the code for testing**
   The tester simply replaces all private members in the source code with public members, which can be accessed by the test framework. The disadvantage of this solution is that it means maintaining two versions of the same implementation.
2. **Using language features**
   Some languages support exclusive access. For example, we can define friend classes in C++ that have privileged access to a given class, here the tested class. The disadvantage of this solution is the dependency of the class under test and the test framework class. The class under test cannot be compiled without the friend class. Moreover, the test concern still cross-cuts the application concern.

We propose an aspect-oriented solution to the encapsulation problem. AspectJ, the aspect-oriented language used in *FlexTest*, provides the keyword `privileged` for aspects, which allow privileged access to the adapted classes. In the *FlexTest* framework, privileged aspects are used in addition to test classes of the object-oriented part.

We show the privileged access using result checking as an example. Test aspects select results of test cases via point-cut definitions. To make definitions of point-cuts easier, they should be defined per object. This means each creation of an object causes the creation of a test aspect, whose point-cuts select only join points of the object's class.

```
1 public privileged aspect TestVectorDouble4Aspect extends TestAspect
2 {
3   public pointcut initMe() :
4       within(TestVectorDouble4);
5   [...]
6 }
```

Fig. 3. Aspect `TestVectorDouble4Aspect` (AOF) supporting class `TestVectorDouble4` (OOF)

Using *FlexTest*, the tester must define a subaspect of the aspect `TestAspect` (AOF) and activate it. This is done in two steps:

1. For each class under test, the tester must define a test case class and an aspect adapting the test case class. The aspect class inherits the abstract

aspect `TestAspect` (AOF) and must override the abstract point-cut `initMe` (cf. line 3 in Fig. 3). In Fig. 3, a fragment of the implementation of the aspect `TestVectorDouble4Aspect` (AOF) is shown, adapting the test case class `TestVectorDouble4` (OOF) (see line 1). The aspect `TestVectorDouble4-Aspect` (AOF) has privileged access to its adapted class and to the objects of the IUT defined in this class.

2. The test case class is inherited from the abstract class `TestCase` (OOF). The test case class, in the example the class `TestVectorDouble4` (OOF), must call `initTestAspect` to activate the aspect, in the example `TestVector-Double4Aspect` (AOF).

```
1  public void testConstructor()
2  {
3    test1 = new VectorDouble4(1., 2., 3., 4.);
4    [...]
5  }
```

Fig. 4. Constructor test in `TestVectorDouble4` (OOF)

Fig. 4 shows an example of a constructor test. The members of `VectorDouble4` (IUT) are encapsulated. They cannot be accessed directly.

```
1  pointcut vector4ConstructorTest(double nX, double nY,
2                                  double nZ, double nW) :
3  call (VectorDouble4.new(double, double, double, double))
4  && args(nX, nY, nZ, nW);
5
6  after(double fX, double fY, double fZ, double fW)
7    returning(VectorDouble4 newConst) :
8      vector4ConstructorTest(fX, fY, fZ, fW)
9  {
10   check(((newConst.m_fX == fX) && (newConst.m_fY == fY)
11      &&  (newConst.m_fZ == fZ) && (newConst.m_fW== fW)));
12 }
```

Fig. 5. Aspect-oriented result check in `TestVectorDouble4Aspect` (AOF)

To gain access to the members of `VectorDouble4` (IUT), we define a point-cut for the construction of `VectorDouble4` (IUT) objects and check the results within an `after` advice (see Fig. 5, lines 3 and 6-8, respectively).

4.2 Quantification of test cases

A popular definition of aspect-oriented programming (AOP) [5] states that "AOP is quantification and obliviousness". In this section, we examine how quantification can be used to improve the test process.

Testing can be a tedious task. It involves writing lots of test cases. Often, test cases can be checked in a similar way, because they return similar results.

AOP quantification can express conditions that must hold for a set of method calls in a certain test case. Grouping test cases gives them a context, which allows an AOP language to check results of method calls in one advice.

```
1   public void testIsNormalPositive()
2   {
3     VectorDouble4 vec2 = new VectorDouble4(0, 1, 0, 0);
4     vec2.isNormal();
5     VectorDouble4 vec3= new VectorDouble4(1, 0, 0, 0);
6     vec3.isNormal();
7     [...]
8   }
9
10  public void testIsNormalNegative()
11  {
12    VectorDouble4 vec = new VectorDouble4(3, 2, 1, 5);
13    vec.isNormal();
14    [...]
15  }
```

Fig. 6. `isNormal` calls in TestVectorDouble4 (OOF)

In Fig. 6, the method `isNormal` is tested returning `true` if a vector is normal and `false` otherwise. We define two contexts. In the first context, `testIsNormalPositive`, we expect all method calls to return `true` (lines 1-8). The second context groups all method calls returning `false` (lines 10-15).

The results are selected by a point-cut defined using the `cflowbelow` statement to select the context (see Fig. 7, lines 2-4).

Above, we describe the use of aspect-oriented programming as test oracle, but it can also serve as test driver. Experiments have shown that other separations of responsibilities are possible using the `FlexTest` framework. For example, we can encapsulate all calls of `isNormal` in an aspect instead of implementing them in the test case class. The next section shows another example the use of aspects as a test driver.

```
1   [...]
2   pointcut vectorDouble4PositiveIsNormalTest() :
3     call (* VectorDouble4.isNormal())
4     && cflowbelow(call(* TestVectorDouble4.testIsNormalPositive()));
5
6   after() returning(boolean retValue) :
7     vectorDouble4PositiveIsNormalTest()
8   {
9     if (!retValue) makeError(thisJoinPoint);
10  }
```

Fig. 7. Testing the results of `isNormal` calls with the expected result `true`

4.3 Testing object hierarchies

In [6], subtyping is defined as follows: "If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T."

Type hierarchies in which subtype relations confirm to this criteria are called Liskov-conform.

To check classes for Liskov-conformity, we have to run superclass test cases on all subclasses, as proposed in [2]. Liskov-conform subclasses should pass all test cases of their superclasses.

Designing test cases for subclasses raises the problem of considering test cases of all superclasses. We can say that the concern of testing for Liskov-conformity interferes with the concern of testing a single class.

Our solution encapsulates the object hierarchy test in one aspect. Thus, the concern of testing for Liskov-conformity is separated from the concern of testing a single subclass. New classes can be added to the hierarchy without changing the hierarchy test module. Our solution selects the relevant test suites for each class and the relevant classes for each test suite itself.

As mentioned earlier, AOP can be understood as quantification and obliviousness. For this task, we need quantification to select the classes under test. This selection must occur without the help of individual test classes and aspects. Thus, the aspect-oriented part of the framework must be more active than in the previous sections. It should not only be triggered to check the results of certain test cases but acts as a test driver that is responsible for running a test suite on a hierarchy of classes. The first instantiation of an object of the superclass or one of its subclasses is intercepted to perform the superclass test suite.

For example, if we wish to test the method `toString` of all `Vector` (IUT) subclasses we must apply the method `testToString`, given in Fig. 8, to all subclasses of `Vector` (IUT). The object `_vecExample` (lines 4, 6, 8) is the actual object under test.

```
1  void testToString()
2  {
3    String strCompare = new String("");
4    for (int i = 0; i < _vecExample.getNoOfElements(); i++)
5    {
6      strCompare += _vecExample.getElement(i);
7    }
8    check(strCompare.equals(_vecExample.toString()));
9  }
```

Fig. 8. Testing the `toString()`-method for all sub-classes of `Vector` (IUT)

```
1 public abstract aspect TestCompleteInheritanceAspect
2   extends TestAspectEntity percflow(newInstance())
3 {
4   public pointcut newInstance() :
5     call (* TestSuite.run());
6   [...]
7 }
```

Fig. 9. Point-cut selecting join point to create a new instance of a hierarchy test aspect

All hierarchy test aspects in `FlexTest` are subaspects of `TestCompleteInheritanceAspect` (AOF). An instance of this subaspect is assigned to each run of a test suite. This is achieved by the code in Fig. 9 (lines 4+5).

```
1 public aspect TestVectorAspect extends TestCompleteInheritanceAspect
2 {
3   public pointcut newSubclass():
4     call (Vector+.new()) && within (TestVector*);
5   [...]
6 }
```

Fig. 10. Point-cut selecting the creation of a new class under test

We must now specify which subclasses should be tested by the hierarchy test aspect. This is done by overriding the abstract point-cut `newSubclass` (line 3 in Fig. 10), so that it selects the creation of certain subclasses in a test context.

Creations of subclass objects of `Vector` (IUT) that are located in classes of a name starting with `TestVector` (OOF) are addressed by the point-cut in Fig. 10 (lines 3+4). `TestCompleteInheritanceAspect` (AOF) subaspects memorize

which classes they have already tested. Each class is tested only once for each run of the test suite.

4.4 Convenience functionality

This section covers issues concerning the organization of a test framework and its context. Some of these can be very neatly implemented using aspect-oriented programming techniques.

Logging. Testing must be documented. A test framework must therefore offer functionality to log the results of test runs following a certain pattern. For example, the execution of all methods that are subclasses of class `TestCase` (OOF) or all results of methods with the name pattern `test*` should be logged. Flexibility is needed in the definition of logging.

Since logging is a classical example of the use of AOP, it is supported by our test framework as well. Instead of scattering the logging calls over the code, they are defined in a central module.

Failure Localization. It is not sufficient for a test framework to merely state that a failure has occurred. A test framework must also locate which class and which source code line have caused the failure.

JUnit [3] throws an exception and analyzes the stack traces for failure localization. In our opinion, this is not a good solution because it abuses the exception mechanism of Java. Exceptions should be thrown when a routine does not know how to handle a situation and passes the responsibility to its caller. A failed test case is normal behaviour in a test framework and using exceptions conflicts with the application's exceptions. This can lead to confusion with exceptions originating from the application under test or with exceptions thrown in the test framework.

The straight modular AOP variant is used to define a point-cut for all methods that check conditions for the test suite and create error strings in their advices. These advices are able to reason about the point-cut shadow and even know the line of code.

4.5 Other applications

Finally, we present two applications of AOP in the test framework that do not specifically focus on testing. Assertions are part of quality assurance, and mock objects are needed in unit testing in some situations.

Mock objects. Mock objects [7] mimic objects without having much functionality themselves. Their purpose is to check if the object under test communicates in the right way with objects of the mimicked type. The use of mock objects may be necessary, if the simulated object is not available at the time of testing or if it

takes to much time to test with all functionality of the mimicked object. Unlike to stubs, mock objects are not just the original classes stripped of functionality. They must provide some means of determining if other objects communicate with the mimicked object in the right way.

Aspects can be mock objects without interfering with the code of the class under test as follows:

- Each mock object is implemented by one aspect.
- The aspect contains a point-cut for each method of the mimicked class. Each point-cut selects the call of one of its methods in the test context.
- The point-cuts are advised by `around` advices without a `proceed` statement.
- The mock object advice checks if it is called at the right time in the right state with the right parameters. Otherwise, it logs an error.

The aspect-oriented mock object approach is especially useful if the insertion of the mock object into a context is difficult. This can be the case if the object to be mocked is, for example, determined in the method under test so that the testing class is not able to influence this object or replace it with a mock object. [8] discusses this problem at greater length.

Assertions. Assertions [2] involving pre- and postconditions are easy to implement in object-oriented programming languages at the beginning and the end of each method. Class invariants, however, are hard to implement. They must hold before and after methods of the actual object are called from other objects. They are therefore scattered across the code if implemented conventionally.

Points at which the class invariant must hold are potential join points in most AOP languages. Thus, inserting class invariants is a typical application for aspect-oriented programming. An aspect selects all method calls of a class via point-cuts and checks the class invariant for the class. With the `cflowbelow`-statement, it is even possible to distinguish calls of other objects from those of the actual object for which the invariant does not necessarily have to hold.

5 Related Work

A couple of research papers have investigated testing using aspect-oriented programming techniques. The non-invasive integration of test code into the system under test has been shown to be the main advantage here.

Most of the work in this area focuses on the popular language AspectJ. Although our framework, too, is based on AspectJ, we see advantages in using other aspect-oriented languages or platforms, like *abc* [9] that provides more join point flexibility, or Object Teams that allows us to explicitly activate and deactivate aspects at run-time. There is, however, no known work based on *abc*. An instrumentation technique using Object Teams for integrating state-based test oracles into the system under test is shown in [10]. The implementation of state-based test oracles with AspectJ is shown in [11].

Aspect-oriented programming techniques are also used in [12–15]. The main focus of these approaches is monitoring run-time behaviour on the system level. Our approach concentrates on unit testing of methods, which means it has a completely different granularity.

[8] proposes integrating mock objects using AspectJ. The work is based on the JUnit test framework and has many similarities with our own work but we have also considered many other applications where aspects can be helpful in testing.

The realization of assertions with aspect-oriented programming techniques is also proposed in [16–18]. In all cases, assertions are defined using a specification language (OCL or JML). For example, in [18] JML specifications are transformed into AspectJ code. In our approach, the tester defines assertions and test cases using advices. The advantage here is that the tester does not have to learn a new language in addition to AspectJ. However, directly written advice code is less expressive than a specification language like JML.

In contrast to the presented related work, our approach is not confined to one test application. We have investigated the use of aspect-oriented programming techniques in different applications, resulting in a flexible test framework for unit testing object-oriented systems.

6 Discussion

Our experiments with the *FlexTest* framework have shown that aspect-oriented programming techniques are suitable for unit testing. Developing a new framework instead of using JUnit enables us to enhance the framework flexibly with aspect-oriented features. Our framework provides some solutions to typical unit test problems.

We now go on to suggest some entry points for discussion and indicate a potential direction for further research in this area.

First, we consider some disadvantages of our approach. As an implementation language for the aspect-oriented part of the framework, AspectJ requires recompilation of the whole application with all test aspects. This adds extra time to the compile process. Furthermore, each change of test aspects leads to a recompilation of the whole application. But this is a specific AspectJ problem. Other aspect-oriented languages that support load-time or run-time weaving only require the compilation of test aspects.

Another disadvantage is due to the tester's experience with aspect-oriented programming. A tester who has no experience with AOP cannot use the *FlexTest* framework.

The separation of the `TestCase` class and the `TestAspect` aspect distributes test implementations to different modules. This seems to be a disadvantage, but in our presented examples the separation is clear between test cases (`TestCase`) and test oracle (`TestAspect`), and it naturally fits with mock object implementation. `FlexTest` supports the strict separation ot testing concern and application

concern. There is no need for a preparation of the classes under test. Also, sub-concerns are separated by the use of AOP:

- Logging and failure localization are separated from testing.
- The concern of testing for Liskov-conformity is separated from the concern of testing each individual class.

We believe that this kind of modularity helps in managing the complexity in large test suites.

FlexTest provides solutions to some common test problems encountered when unit testing object-oriented applications, like encapsulation bypassing. Aspect-oriented programming provides a flexible instrumentation solution that can be easily extended to other instrumentation problems. One of the main drawbacks - the lack of instrumentation support per source-code statement - is compensated for by the flexible and entensible join point language provided by the *abc* compiler [9]. The *abc* compiler allows us to enhance the join point language so it even supports instrumentation per statement.

Owing the modular structure of the *FlexTest* framework, it is easy to customize the logging and failure localization functionality for test suites. To log in a different manner or change the failure localization feature, one merely has to edit few lines in a central aspect.

The pros and cons of our approach show that the main problem is the difficulty of expressing the functionality in an AOP language in an easy understandable manner. The complex expressions to select test method calls should be replaced by expressions that are easier to read and understand. This means that we have to change the AOP language.
Future work involves considering the following issues:

- Comparing the FlexTest framework with an object-oriented framework like JUnit in a case study.
- Defining new keywords for use in test situations so that point-cut selections are easier to read.
- Extending the point-cut language to a point where it is possible to check loop invariants.
- Integrating the test concern into a language in which test classes and test aspects are more closely related.

The new AOP language could be based on the extensible compiler *abc*, which implements full AspectJ support but provides a flexible join point language.

References

1. Vösgen, M.: FlexTest Framework. http://swt.cs.tu-berlin.de/ mvoesgen/Stuff/flextest.zip (2005)
2. Binder, R.V.: Testing Object-Oriented Systems. Object Technology Series. Addison-Wesley (1999)
3. JUnit: JUnit-Homepage. http://www.junit.org (2005)

4. AspectJ: AspectJ-Homepage. http://www.aspectj.org (2005)
5. Filman, R.E., Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness. In: Workshop on Advanced Separation of Concerns, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Minneapolis, Minnesota, USA (2000)
6. Liskov, B.: Data Abstraction and Hierarchy. In: Addendum to the proceedings on Object-oriented programming systems, languages and applications, Orlando, Florida, USA (1987) 17 – 34
7. Mackinnon, T., Freeman, S., Craig, P.: EndoTesting: Unit Testing with Mock Objects. In: eXtreme Programming and Flexible Processes in Software Engineering (XP), Cagliari, Italy (2000)
8. Lesiecki, N.: Test Flexibility with AspectJ and Mock Objects. http://www-106.ibm.com/developerworks/java/library/j-aspectj2/ (2002)
9. Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: abc: An Extensible AspectJ Compiler. In: International Conference on Aspect-Oriented Software Development (AOSD), Chicago, Illinois, USA (2005)
10. Sokenou, D., Herrmann, S.: Using Object Teams for State-Based Class Testing. Technical report, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Berlin, Germany (2004)
11. Bruel, J.M., Araújo, J., Moreira, A., Royer, A.: Using Aspects to Develop Built-In Tests for Components. In: AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML), San Francisco, California, USA (2003)
12. Deters, M., Cytron, R.K.: Introduction of Program Instrumentation using Aspects. In: Workshop of Advanced Separation of Concerns in Object-Oriented Systems, 16th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM Sigplan Notices, Tampa, Florida, USA (2001)
13. Filman, R.E., Havelund, K.: Source-Code Instrumentation and Quantification of Events. In: Workshop on Foundations of Aspect-Oriented Languages, 1st Intern ational Conference on Aspect-Oriented Software Development (AOSD), Enschede, Netherlands (2002)
14. Low, T.: Designing, Modelling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). In: Workshop on Aspect-Oriented Modeling with UML, 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, Netherlands (2002)
15. Mahrenholz, D., Spinczyk, O., Schröder-Preikschat, W.: Program Instrumentation for Debugging and Monitoring with AspectC++. In: Proceedings of The 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC), Crystal City, Virginia, USA (2002)
16. Richters, M., Gogolla, M.: Aspect-Oriented Monitoring of UML and OCL Constraints. In: AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML), San Francisco, California, USA (2003)
17. Briand, L.C., Dzidek, W., Labiche, Y.: Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Technical report, Carlton University, Ottawa, Canada (2004)
18. Xu, G., Yang, Z., Huang, H.: A Basic Model for Aspect-Oriented Unit Testing. www.cs.ecnu.edu.cn/sel/ harryxu/research/ papers/fates04_aspect-oriented