Incremental Modular Testing for AOP

André Restivo[1], Ademar Aguiar[1] and Ana Moreira[2]
[1]*Faculdade de Engenharia da Universidade do Porto, Porto, Portugal*
[2]*NOVA LINCS, Universidade Nova de Lisboa, Costa da Caparica, Portugal*
{arestivo, aaguiar}@fe.up.pt, amm@fct.unl.pt

Keywords: Testing, Aspects, Modularity.

Abstract: By designing systems as sets of modules that can be composed into larger applications, developers unleash a multitude of advantages. The promise of AOP (Aspect-Oriented Programming) is to enable developers to organize crosscutting concerns into separate units of modularity making it easier to accomplish this vision. However, AOP does not allow unit tests to be untangled, which impairs the development of properly tested independent modules. This paper presents a technique that enables developers to encapsulate crosscutting concerns using AOP and still be able to develop reusable unit tests. Our approach uses incremental testing and invasive aspects to modify and adapt tests. The approach was evaluated in a medium scale project with promising results. Without using the proposed technique, due to the presence of invasive aspects, some unit tests would have to be discarded or modified to accommodate the changes made by them. This would have a profound impact on the overall modularity and, in particular, on the reusability of those modules. We will show that this technique enables proper unit tests that can be reused even when coupled with aspect-oriented code.

1 INTRODUCTION

The development of large software projects is a complex task. Unfortunately, humans often struggle when asked to cope with complex problems. The way we usually deal with this is by decomposing the larger problem into several smaller and more manageable ones. When talking about actual code, we usually call these smaller pieces of software *modules*.

To reap as many advantages as possible from this division into smaller modules, several important aspects should be taken into consideration. Modules should have low coupling and high cohesion between them and concerns should not be spread over several modules or tangled inside one. A good decomposition should lead to modules that can be described, reused, replaced, and tested in isolation.

Classical paradigms, like Object Oriented Programming (OOP), suffer from the *tyranny of the dominant decomposition* (Peri Tarr et al., 1999) that states that when programs are modularized following any given decomposition criteria, all the concerns that do not align with that criteria end up tangled and scattered throughout several modules of the system. Aspect-Oriented Programming (AOP) aims at encapsulating these crosscutting concerns into separate units of modularity (Gregor Kiczales et al., 1997).

One way AOP languages, like *AspectJ* (The Eclipse Foundation, 2010), bypass this limitation is by allowing programmers to isolate these crosscutting concerns into separate units, called *aspects*. An *aspect* defines a set of advices that run whenever a set of selected points (*joinpoints*) in the underlying system is reached. This allows units containing aspects to change the behaviour of other units.

Aspect-oriented software development promises to enable developers to achieve this kind of isolation, not only at the code level, but also in the requirements (Rashid et al., 2003) and design (Baniassad and Clarke, 2004) phases. By using aspects, developers are able to separate each concern into its own unit of modularity. Having concerns untangled improves reusability as the code of each module pertains only to a single concern.

However, when modules are reused, there are other artifacts besides the actual code that must be transferred between projects. Some of those artifacts are the tests that help develop reliable software.

In this paper, we argue that due to the nature of aspects, some unit tests cannot be reused in different contexts thus impeding module reusability. Testing modules in isolation also becomes harder. We will present a technique that uses automatic dependency detection and incremental compilation, together with

Restivo, A., Aguiar, A. and Moreira, A.
Incremental Modular Testing for AOP.
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT*, pages 50-59
ISBN: 978-989-758-194-6

some *Java* annotations, that allows the cohexistance of testing methodologies together with AOP code while still keeping the code portable. To support this technique, a tool has also been developed and will be presented in this paper.

In Section 2, the problem we propose to tackle will be identified. Section 3 describes a technique, based on incremental testing, that aims at solving the proposed problem. Section 4 presents a *Eclipse* based implementation of the technique. Our efforts to validate the solution are presented in Section 5. Finally, Sections 6 and 7 describe related work and our conclusions.

2 THE PROBLEM

Let us start with a simple example in *AspectJ*. Imagine we have a base class called *Question* that represents a multiple choice question in an exam (see Listing 1 for a simplified sample of the class).

Listing 1: Question Class

```
public class Question {
  public void addChoice (String choice)
    {...}
  public String getChoice(int number)
    {...}
}
```

This class is part of a module with the same name and has several tests. One of these tests adds some choices to the question and verifies if they are present and in the correct order (see Listing 2 for a simplified version of the test).

Listing 2: Question Test.

```
public void testChoices() {
  Question q =
    new Question("Choose a color?");
  q.addChoice("blue");
  q.addChoice("red");
  q.addChoice("green");

  assertEquals("blue", q.getChoice(0));
  assertEquals("red", q.getChoice(1));
  assertEquals("green", q.getChoice(2))
      ;
}
```

As we have postulated before, this test should also be reusable. If the *Question* module is reused in another project, it should also be possible to reuse the test in that project without any modifications. In fact, both test and code should be seen as parts of a single reusable artefact.

In a classical object-oriented environment, this would not be a problem. This class and its test would always work in the same way regardless of any other artifacts present in the system. This does not happen when aspects are present. Imagine that we add an aspect called *RandomizeChoices* that changes the positions of the choices at random as they are added (see Listing 3 for a small excerpt of this aspect).

Listing 3: Randomize Aspect.

```
pointcut addChoice(List list,
                   String choice) :
  cflow(
    call(void Question.addChoice(..))
  ) &&
  !within(Randomize) &&
  target(list) && args(choice) &&
  call(boolean List.add(..));

boolean around(List list,
               String choice) :
  addChoice(list, choice) {
   int position =
      random.nextInt(list.size() + 1);
   list.add(position, choice);
   return true;
}
```

When this aspect is added, our *testChoices* test will start failing five times out of six. We did not change the code of the *Question* class or any code this class depends on. In object-oriented programming, the problem of having code from outside a module influencing the outcome of a test was solved by using *mocks* and *stubs* (Fowler, 2007). The difference is that, when dealing with objects, only the code the module depends on can alter its behavior. As we just saw, that is not the case when coding with aspects. In the next paragraphs we will describe some naive solutions for this problem.

Moving the Test. The most simple solution would be to move the offending test from the *Question* module to the *RandomChoices* module and change it to accommodate the new concern. This could be easily achieved by using the *contains* method to test for the presence of a choice instead of looking for it in the expected location. The problem with this approach is that the *Question* module would lose the *testChoices* test. This would make it harder to reuse this module in other systems. At the same time, the basic functionality of being able to add choices to questions would no longer be tested separately.

Changing the Test. By altering the *testChoices* test to accommodate the changes introduced by the *RandomChoices* aspect, we could easily make it work again. This is the same solution as the one we saw before but instead of moving the test to the other module and making the changes there, we just change the test we already have. This would also have the effect of

making the *Question* and *RandomChoices* concerns tangled with each other – not at the working code level but at the testing level – preventing the *Question* module from being easily reused and leaving the *RandomChoices* without any tests.

Using Aspects to Change the Test. We could keep the *Question* module code unchanged and use an aspect to change the *testChoices* test behavior whenever the *RandomChoice* module is present in the system. This way the *Question* tests would work as planned when the module is used in isolation and the *RandomChoices* module would have a different test for its own behavior. Although having some advantages, the problem with this approach is the same as in the previous one. The difference is that the tangling now happens in the *RandomChoices* module.

None of these solutions gives us a scenario where modularity is preserved in its entirety. However, we could summarize the principles in what would be a good solution:

1. **Obliviousness.** Tests should only test the behavior of their own modules.
2. **Completeness.** All concerns should have their own tests and all tests should run at least once.
3. **Correctness.** When a module is reused in a different context, tests should still work correctly.

In the next section, we will describe a technique based on incremental compilation that allows the usage of unit tests without breaking modularity. For the sake of completeness, we will explore four different ideas that will converge into our final proposition and we will explain how these compare to one another in several different aspects.

3 INCREMENTAL TESTING

A well-designed software system should be built in such a way that low-level modules do not depend on higher level modules. Software systems should be built layer by layer, with each layer adding more functionalities. If this is accomplished, then the modules dependency graph becomes a *directed acyclic graph* (DAG).

Unfortunately, not all software systems follow this recommendation and it is common to find circular dependencies even in well-designed software systems. In graph theory, these collection of nodes that form circular dependencies are called *strongly connected components*, and although they cannot be easily removed, they can be isolated. We do this by considering each *strongly connected component* in the graph as a super module. In this way, we can consider that all software systems can be thought of as being composed as a DAG of module dependencies.

If we are able to extract this dependency graph from the source code, we can be sure that we will have at least one low-level module, lets call it module *A*, that does not depend upon any other module. This module can be compiled and tested in isolation shielding it from the potential influence of higher level aspects.

After this initial module is tested, we can take another module that only depends on this module, lets call it module *B* and test both of them together. Tests from module *B* can be easily shielded from eventual errors in the source code of module *A* by using classic object-oriented unit testing techniques like *mocks* and *stubs*. On the other hand, tests from module *A* can still be influenced by aspects on module *B* making them fail. We already ran the tests of module *A* once, so all we need to do is to make sure that tests that fail under the influence of aspects from module *B* are not run again. This process can then be repeated for every module in the system until all tests have run at least once.

When we first started researching this idea (Restivo and Aguiar, 2008), we postulated that tests could be used to detect unexpected interferences caused by aspects. An unexpected interference happens when a module containing invasive aspectual code changes the behavior of another module in unforeseen and undesirable ways.

If tests are in place, these interferences can be easily detected. However, there will be no apparent difference between an unexpected interference and an expected interaction. The developer must be able to differentiate between the two of them and act accordingly. Interferences must be fixed and interactions must be dealt with; not because they are wrong but because they impede the testing process.

To fix the testing process, the developer must be able to specify that a certain interaction is desirable. After that, the testing process can ignore any tests that fail due to that interaction but only after the test has been successfully executed without the offending aspect. In our initial approach, we considered using code annotations to enable the developer to specify these interactions. In the following sections, we will demonstrate how that initial approach evolved and present the advantages and drawbacks of each step.

3.1 Method-Test Approach

Our initial idea was to consider tests as being the proof that a certain concern was implemented correctly. Ideally, for every concern in the system, the

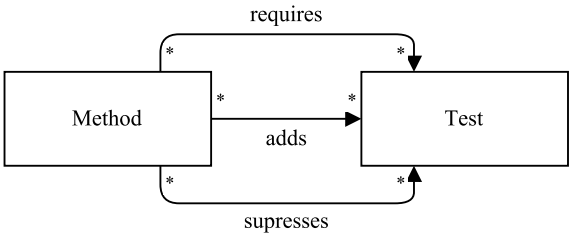

Figure 1: Method-Test approach.

developer should be able to create a test for it. Class methods are the artefacts that end up implementing those concerns. So we could have annotations in each method with a reference to the test for the concern that the method was implementing.

In order to create the DAG of dependencies needed for our incremental testing process, we proposed another annotation where each method could declare the tests it depends on. Notice that we do not specify which methods the method depends on, but the tests that were created to test that method. This means that every time an aspect is added to the system, in our incremental testing process, and a previously tested test fails, we can pinpoint which methods are affected by that interaction.

Finally, we proposed another annotation that allows developers to declare expected interactions. This annotation would be used by developers on advices to pinpoint which tests they expect to break. Figure 1 contains a representation of the connections achieved by these annotations. Listing 4 represents a sample of the code needed to implement this approach for the example described in the beginning of this paper.

Listing 4: Method-Test Approach.

```
public class Question {
  @Adds("QuestionTest.testChoices")
  public void addChoice (String choice)
    {...}
  @Adds("QuestionTest.testChoices")
  public String getChoice(int number)
    {...}
}

/* Inside Question Test Suite */
public void testChoices() {...};

/* Inside Randomize Aspect */
@Removes("QuestionTest.testChoices")
boolean around(List list,
               String choice) :
  addChoice(list, choice) {...};
```

Having these annotations in place, our testing process would be able to create the DAG of dependencies, using the *requires* and *adds* annotations, and run only the tests that have not been removed by subsequent advices by means of the *removes* annotation.

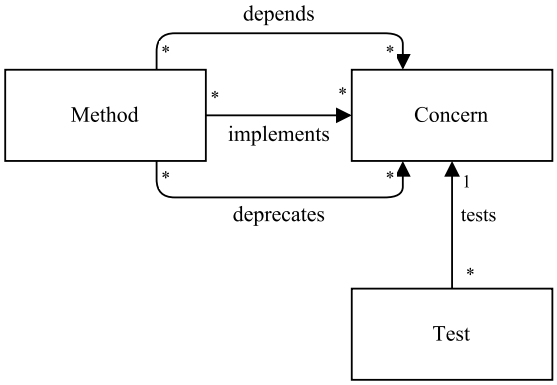

Figure 2: Concern-Test approach.

Besides the extra effort put in by the developer, the problem with this approach is that the relation between methods/advices and tests is artificial.

3.2 Concern-Test Approach

To mitigate the artificiality of our first approach, we decided to add a new annotation that would depict a *concern*. In this approach, each method has an annotation stating which concern it implements. These concerns can even be derived from the requirements phase.

In this way, methods and advices no longer add, remove or depend on tests but on concerns. To know which concern is tested by each test we need an annotation that will be applied to each test with a reference to the concern. Figure 2 shows the relationships between the code artifacts derived from the annotations in the code. Listing 5 represents a sample of the code needed to implement this approach for the example described in the beggining of this paper.

Listing 5: Concern-Test Approach

```
public class Question {
  @Implements("questionHasChoices")
  public void addChoice (String choice)
       {...}
  @Implements("questionHasChoices")
  public String getChoice(int number)
      {...}
}

/* Inside Question Test Suite */
@Tests("Question.questionHasChoices")
public void testChoices() {...};

/* Inside Randomize Aspect */
@Removes("Question.questionHasChoices")
boolean around(List list,
               String choice) :
  addChoice(list, choice) {...}
```

To apply our proposed testing process using this

approach, we start by selecting a module whose methods do not depend on any concern from another module. Tests for the concerns defined in the module are run. In each step we add another module that only has *requires* annotations referencing concerns added by modules that already have been tested. If a test that passed in a previous step fails after a new module is added we can infer that there is an interaction between a concern implemented in that module and the concern that the failing test was testing.

In comparison with the first approach, this one has a richer set of metadata on the implemented concerns and their tests. This extra knowledge allows us to better understand which concerns are interacting with each other. Developers can therefore reason more easily if the interaction is expected or if it is an unexpected interference.

3.3 Module-Test Approach

The previous two approaches imposed an heavy burden on the developers as they had to add a lot of annotations to the code. In this iteration we tried to reduce the amount of extra work needed by removing most of them.

We started by considering modules as being defined by the way the used language, in this case *AspectJ*, defined its own units of modularity – *Java* packages. To prevent cases where the relation between the language defined units and the intended modules is not a direct one, we added an optional annotation so that each class/aspect could define to which module it belongs.

Tests defined inside a module are considered as being used to test some concern of the module. This removed the burden to add annotations for each test.

The only annotations really needed, are between tests. The *replaces* annotations identify cases where a test represents a concern, developed as an invasive aspect, that changes the behavior of another concern that is tested by the other test. Figure 3 shows the relationships between the code artifacts derived from the annotations in the code. Listing 6 represents a sample of the code needed to implement this approach for the example described in the beginning of this paper.

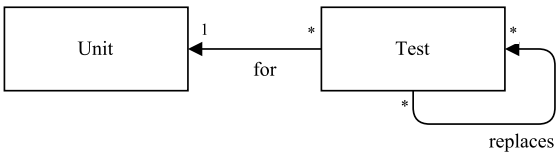

Figure 3: Module-Test approach.

Listing 6: Module-Test Approach.

```
public class Question {
  public void addChoice (String choice)
    {...}
  public String getChoice(int number)
    {...}
}

/* Inside Question Test Suite */
public void testChoices() {...};

/* Inside Randomize Aspect */
boolean around(List list, String choice
    ) :
  addChoice(list, choice) {...}

/* Inside Randomize Test Suite */
@Replaces("Question.QuestionTest.
    testChoices")
public void testRandomChoices() {...};
```

This approach drastically reduced the amount of extra work by the developer. However, the information gathered is much less. But still, when interactions are detected we can get information about which test failed and which modules caused the interaction. This information should be enough for the developer to identify the origin of the problem and act accordingly.

3.4 Advice-Test Approach

The last approach considered was an easy evolution from the previous one. The only mandatory annotation in our previous approach was used to remove a test from the system when a module containing invasive aspects was added to the system changing the behavior the test was testing.

An alternative would be to use an advice to disable the test. Listing 7 shows how that can be accomplished by simply adding an around advice that does not call the original captured joinpoint from the test.

Listing 7: Module-Test Approach.

```
public class Question {
  public void addChoice (String choice)
      {...}
  public String getChoice(int number)
     {...}
}

/* Inside Question Test Suite */
public void testChoices() {...};

/* Inside Randomize Aspect */
boolean around(List list, String choice
    ) :
  addChoice(list, choice) {...}
void around() :
```

```
    testChoices() {}

/* Inside Randomize Test Suite */
public void testRandomChoices() {...};
```

Although this approach does not use any annotations, besides the optional one that changes the way modules are defined as language constructs, the incremental compilation process is still needed to ensure that disabled tests are run at least once during testing.

3.5 The Process

The process used for all these approached is, in its essence, the same:

1. Identify all modules, their tests and dependencies.
2. Execute a *strongly connected* analysis of the dependency graph transforming it into a DAG of modules.
3. Execute a *topological sort* to determine in which order the components must be compiled.
4. For each component:
   (a) Compile it together with the previously tested components.
   (b) Execute the tests defined for the features provided by this component.
   (c) Execute the tests defined by previous components.

When a test fails in step 4b, it means that the module being added to the system has an error. This error can either be caused by a test not working properly, an error in the code of this module, or even a problem related to errors in the previously compiled modules that was not detected by the implemented tests.

When a test fails in step 4c, it means we have encountered an interaction between the code of the module being tested and one of the modules previously compiled. Depending on the selected approach, the information given to the developer can be different. If using the *Concern-Test*, it should be possible to pinpoint the concern that is being interfered with. The other approaches would only reveal the test being broken.

4 IMPLEMENTATION

During the course of the work, two different plugins were developed: *DrUID* and *Aida*. Both are based on the usage of annotations throughout the code that contain information about which interferences are expected. These tools were implemented as *Eclipse* plugins.

AspectJ was chosen as the target language for several reasons. First, it is one of the most used aspect-oriented languages. Secondly, as it is Java based, it can be used with *Eclipse*, an IDE where plugin development is straightforward. With *Eclipse* we also get two other important benefits in the form of the tools JDT and AJD for Java and AspectJ languages respectively. They allow access to the source code abstract syntax tree. Although the implementation is *AspectJ* oriented, the technique we proposed is applicable to other aspect-oriented languages following the same principles.

4.1 DrUID

DrUID (UID as in Unexpected Interference Detection) (Restivo, 2009; Restivo and Aguiar, 2009) was the first attempt at creating a plugin to help developers follow the methodology being explored throughout this paper. In order to accomplish this, the plugin allows developers to define several characteristics about system artifacts using Java annotations.

Several aids have been implemented to guide the developer in this process in the form of Eclipse *quick fixes* and *quick assists*. Each time a file is saved in Eclipse, the annotations are inspected and any errors are reported. Besides that, a dependency graph is created and shown in a graphical form that allows the developer to navigate through the code following the dependencies between artifacts.

4.2 Aida

Aida (Restivo, 2010) is an evolution of the DrUID tool, built from scratch, having the main objective of removing most of the burden put on the developer to annotate his code. It also has a bigger focus on the testing process. In this tool, we started by removing the notion of annotating features manually. We did this by considering each test as a feature. This means that the developer only needs to create test cases for each individual behavior. Obviously, this also removed the need to specify which test case tests what feature.

Using code inspection, we were also able to remove the need of specifying the dependencies between features. At the cost of losing some of the details of the dependency graph used in DrUID, with Aida we rely only on the dependencies between units. In the end, we were down to only two types of annotations:

- **@TestFor** Used to indicate which unit each test is testing.

- **@ReplacesTest** Used to indicate that a test replaces another test. It also indicates that if the unit the test is related to is present in the system, then the replaced test does not have to be run.

Units are defined as being contained inside Java packages by default. A third optional annotation (**@Unit**) can be used to alter this behavior. The dependencies between units are automatically calculated by using the information provided by the JDT and AJDT Eclipse plugins.

With the dependency graph calculated, the test process is very similar to that of *DrUID*. We start by extracting the dependency graph from the source code, then we order the units by sorting them topologically and test them adding each unit incrementally to the system.

After running the complete set of tests, *Aida* is capable of reporting, both graphically and in text, on eventually detected errors and interferences. This allows the developer to add **@ReplaceTest** annotations, when an interaction is expected, or correct his code if the interaction was unexpected.

4.3 Current Issues

There are still some issues with the implementation of these tools. *Aida* has been a major step forward as it removes most of the burden of declaring the dependencies from the developer, but there are still a couple of issues.

The first problem is that not all dependencies can be detected. At the moment, Aida is able to detect dependencies caused by: import declarations, method and constructor calls, type declarations and advices. These encompass most of the cases, but soft dependencies, like the ones created using reflection are not detected.

The second problem is that every time the project is tested, all the tests have to be run again. This problem is augmented by the fact that most tests are being run several times. This problem could be mitigated by doing some code analysis to figure which tests might have their results altered by the introduction of a new unit in the incremental compilation process.

5 VALIDATION

To validate the approach we used it in several small sized projects and a medium sized one. The characteristics that we were looking for in a candidate project were that it had to be developed in *AspectJ*, it had to have few circular dependencies between modules and it had to have a test framework.

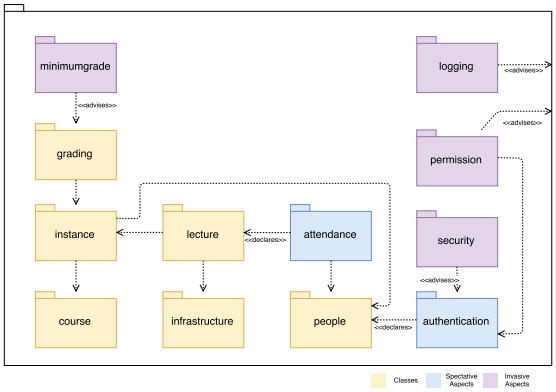

Figure 4: School Testbed Packages.

Unfortunately, all the existing open source projects we considered fail in one of these three aspects. For example, the two most used testbed projects for *AspectJ* are *AJHotDraw* (Marius Marin and van Deursen, 2007) and *Health Watcher* (Greenwood et al., 2007). The first of these has an architecture with a dependency graph so complicated that most of the code is part of a mass of 14 different packages that depend on each other forming a *strongly connected component*. The second one is a much cleaner project, but unfortunately, there are no tests developed for it.

Having failed to elect a good and popular testbed where to run our testing process, we ended up developing our own testbed. A simple school information system (Restivo, 2014) was implemented featuring personal information for students, teacher and administrators, course information, class schedules, infrastructure information and grading. Figure 4 shows the dependencies between the implemented packages.

After implementing the base packages, some packages containing aspects were added to the system:

Authentication. Spectative aspect that adds a login and password attributes to the Person class. Offers methods to login and logoff as well as a way to verify who is logged in.

Attendance. Adds a list of students that attended a certain lecture and methods to manage that list.

Security. Invasive aspect that assures that the passwords are hashed using a secure hashing algorithm. For this, it advises the methods that set and verify passwords of the Authentication module.

Permission. Invasive aspect that verifies that the logged in user has permissions to execute the command being executed. Advises almost every method in the code in order to do this verification.

Logging. Spectative aspect that logs to a file impor-

tant information. At the moment only the creation of new objects and login attempts are logged. To do this, it advises the object creation methods but does not change their behavior.

Minimum Grade. Invasive aspect that adds the possibility of a course evaluation having a minimum grade that the student must attain to pass the course. Adds methods to define this minimum grade and advises the methods that calculate the student final grade.

Each one of the packages in the system was thoroughly tested. The total number of tests amounts to 55 with most of them belonging to the *Permission* package. This happens as this package crosscuts the entire application and modifies the behavior of almost all methods by adding a permission system. This makes it important to test if those methods are still working when the user has permission to use them, and also if access is denied when the user has no permission to use them.

By using *Aida*, interferences were easily spotted. Each time an invasive aspect was added, a test broke somewhere. In the rare event where that did not happen, it was due to an error in the implementation of the new aspect or a poorly written test. By using the technique described in this document, we were able to test all the packages of the system in isolation, without compromising modularity.

After testing the complete system, we tried to test smaller subsets of the system where some packages were not considered. We counted 77 different possible valid configurations with only some of the non-aspectual packages being used. If we add the other four invasive packages, in any possible combination, we get eight times more possibilities. Or a grand total of 616 configurations. We were able to test all of these successfully, using *Aida* without having to add, remove or change any of the tests.

6 RELATED WORK

Katz (Katz, 2004) proposed the use of regression testing and regression verification as tools that could help identifying harmful *aspects*. The idea behind this technique is to use regression testing as normally and then weave each *aspect* into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular *aspect*. This approach is similar to the one presented in this paper but does not explore the possibility of adding the aspects in an automatic and controled order or the extra information that can be extracted by compiling the aspects in different orders. It also does not propose a wat of dealing with intended interactions.

Ceccato (Ceccato et al., 2005) proposed a technique to establish which tests had to be rerun when incrementally adding *aspects* to a system. Combining this technique with our approach could reduce the ammount of time needed to run the tests as some tests would not have to be run twice if it can be proved that they will yield the same result.

Balzarotti (Balzarotti and Monga, 2004) claims that the interaction detection problem can be solved by using a technique proposed in the early 80s, called program slicing. Although totally automatic, this technique does not account for intended interactions.

Havinga (Havinga et al., 2007) proposed a method based on modeling programs as graphs and *aspect* introductions as graph transformation rules. Using these two models it is then possible to detect conflicts caused by *aspect* introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code. Although interesting, this approach suffers the same problem of other automatic approaches to this problem, as intentional interactions cannot be differentiated from unintentional ones.

Lagaisse (Lagaisse et al., 2004) proposed an extension to the Design by Contract paradigm by allowing *aspects* to define what they expect of the system and how they will change it. This will allow the detection of interactions by other *aspects* that were weaved before, as well as the detection of interactions by *aspects* that are bounded to be weaved later in the process.

It has been noticed by Kienzle (Kienzle et al., 2003) that *aspects* can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each *aspect* it would be possible to detect interactions (for example, an *aspect* that removes a service needed by another *aspect*) and to choose better weaving orders.

A state-based testing method for aspect-oriented software has been developed by Silveira (Silveira et al., 2014). According to the authors, this method provides class–aspect and aspect–aspect faults detecting capabilities.

Assunção (Assunção et al., 2014) explored different ways to determine the order for integration and testing of aspects and classes. Two different strategies, incremental and combined, for integration testing were evaluated.

7 CONCLUSIONS

In this paper, we identified a problem that makes it hard to use unit testing in conjunction with aspect-oriented code. The problem is that unit tests in AOP systems must always test the system after the advises from any modules containing aspects have been applied. If these aspects are invasive, then the tests are not testing the unit in isolation and they stop being unit tests.

The solution we proposed is based on having tests, that test modules that contain invasive aspects, annotated in such a way that they announce which tests test the functionality being modified by those aspects. Having these annotations in place would allow a testing technique based on incremental compilation that could test units in lower layers of the software, using their own unit tests, separately from invasive aspects from higher layers. We argued that this is similar to what *stubs* and *mocks* contributed to object-oriented unit testing.

We do not argue that the proposed solution is usable in every situation, but we have shown that it can be used in several different scenarios. We envision it being used in software houses that have a large repository of modules that can be combined in different ways in order to compose different software solutions. Anyone that has tried to create such a system knows that crosscutting concerns are a big issue.

ACKNOWLEDGEMENTS

We would like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

REFERENCES

Assunção, W. K. G., Colanzi, T. E., Vergilio, S. R., and Ramirez Pozo, A. T. (2014). Evaluating different strategies for integration testing of aspect-oriented programs. *Journal of the Brazilian Computer Society*, 20(1).

Balzarotti, D. and Monga, M. (2004). Using program slicing to analyze aspect-oriented composition. In *Proceedings of Foundations of Aspect-Oriented Languages Workshop at AOSD*, pages 25–29.

Baniassad, E. and Clarke, S. (2004). Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering*, pages 158–167. IEEE Computer Society.

Ceccato, M., Tonella, P., and Ricca, F. (2005). Is AOP Code Easier to Test than OOP Code? In *Workshop on Testing Aspect-Oriented Programs, International Conference on Aspect-Oriented Software Development*, Chicago, Illinois.

Fowler, M. (2007). Mocks aren't stubs. Online article at martinfowler.com `http://bit.ly/18BPLE1`.

Greenwood, P., Garcia, A. F., Bartolomei, T., Soares, S., Borba, P., and Rashid, A. (2007). On the design of an end-to-end aosd testbed for software stability. In *ASAT: Proceedings of the 1st International Workshop on Assessment of Aspect-Oriented Technologies*, Vancouver, Canada. Citeseer.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin (1997). Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1241:220–242.

Havinga, W., Nagy, I., Bergmans, L., and Aksit, M. (2007). A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 85–95, New York, NY, USA. ACM Press.

Katz, S. (2004). Diagnosis of harmful aspects using regression verification. In *FOAL: Foundations Of Aspect-Oriented Languages*, pages 1–6.

Kienzle, J., Yu, Y., and Xiong, J. (2003). On composition and reuse of aspects. In *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*.

Lagaisse, B., Joosen, W., and De Win, B. (2004). Managing semantic interference with aspect integration contracts. In *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*.

Marius Marin, L. M. and van Deursen, A. (2007). An integrated crosscutting concern migration strategy and its application to jhotdraw. Technical report, Delft University of Technology Software Engineering Research Group.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. (1999). N degrees of separation: multi-dimensional separation of concerns. *Proceedings of the International Conference on Software Engineering*, page 107.

Rashid, A., Moreira, A., and Araújo, J. (2003). Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 11–20. ACM.

Restivo, A. (2009). DrUID: Unexpected interactions detection. `https://github.com/arestivo/druid`.

Restivo, A. (2010). Aida: Automatic interference detection for aspectj. `https://github.com/arestivo/aida`.

Restivo, A. (2014). School-aspectj-testbed. `https://github.com/arestivo/School-AspectJ-Testbed`.

Restivo, A. and Aguiar, A. (2008). Disciplined composition of aspects using tests. In *LATE: Proceedings of the 2008 AOSD Workshop on Linking Aspect Technology and Evolution*, LATE '08, pages 8:1–8:5, New York, NY, USA. ACM.

Restivo, A. and Aguiar, A. (2009). DrUID – unexpected

interactions detection. AOSD: Demonstration at the Aspect Oriented Software Development Conference.

Silveira, F. F., da Cunha, A. M., and Lisbôa, M. L. (2014). A state-based testing method for detecting aspect composition faults. In Murgante, B., Misra, S., Rocha, A., Torre, C., Rocha, J., Falcão, M., Taniar, D., Apduhan, B., and Gervasi, O., editors, *Computational Science and Its Applications – ICCSA 2014*, volume 8583 of *Lecture Notes in Computer Science*, pages 418–433. Springer International Publishing.

The Eclipse Foundation (2010). The AspectJ Project. http://www.eclipse.org/aspectj/.