Towards Detecting and Solving Aspect Conflicts and Interferences Using Unit Tests

André Restivo
Faculdade de Engenharia da Universidade do Porto
arestivo@fe.up.pt

Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto,
INESC Porto
aaguiar@fe.up.pt

Abstract

Aspect Oriented Programming (AOP) is a programming paradigm that aims at solving the problem of crosscutting concerns being normally scattered throughout several units of an application.

Although an important step forward in the search for modularity, by breaking the notion of encapsulation introduced by Object Oriented Programming (OOP), AOP has proven to be prone to numerous problems caused by conflicts and interferences between aspects.

This paper presents work that explores the proven unit testing techniques as a mean to help developers describe the behavior of their aspects and to advise them about possible conflicts and interferences.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Design, Languages, Verification

Keywords AOP, Conflicts, Interferences, Unit Testing

1. Introduction

Separation of concerns (SoC) has always been the main goal of software engineering. It refers to the ability to identify, encapsulate, and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose [1].

Aspect-Oriented Programming(AOP) [2] is a new programming paradigm that builds on the success of proved paradigms, like Object-Oriented Programming(OOP). The main idea behind AOP is that concerns crosscutting several modules of an application can be developed as single units of modularity and weaved into the application, through a process of composition, in specific points called joinpoints. The motivation behind AOP is to improve the overall modularity of an application by enabling the ability to develop crosscutting concerns as separate units.

AOP also aims for obliviousness [3], i.e. a developer should not have to know about any other aspects that are being weaved into the application code. To make this a reality, aspects have to be allowed to attach virtually to any position of the original source code. In this way, each developer only has to be concerned about his own modules.

Workshop SPLAT '07 March 12-13, 2007 Vancouver, British Columbia, Canada

However, *obliviousness* has been harder to achieve than expected, as conflicts between aspects are prone to occur in large applications with an heavy usage of AOP. In the OOP world conflicts are avoided by using encapsulation techniques and also by using Unit Testing or the Design by Contract (DbC) [4] approaches. Encapsulation allowed developers to know that changing the inner working of an unit would not break another part of the application as long as their public interfaces would be kept the same. Unit testing and the DbC approaches helped developers detect problems caused by an unit changing its public behavior in an easier way (i.e. Regression Testing).

As most AOP languages allow to weave aspects into the private methods of an unit, thus changing their inner workings, encapsulation is no longer a guarantee. However, this is an important feature of the AOP paradigm that allows crosscutting concerns, such as *logging* and *security*, to be easily added by means of a separate modular unit. The problem is that conflicts will certainly arise due to the new possibilities this brings.

Besides the loss of encapsulation, unit tests and contracts suddenly became harder to use mainly because aspects can change the public behavior of an unit thus making obsolete some unit tests and contracts attached to that unit. Unit testing and the DbC approaches have therefore to be rethought in order to accommodate this new way of programming.

In this position paper we will argue how the unit testing approach can still be effectively used in AOP, retaining the same characteristics that made it a very popular Regression Testing methodology, and also be helpful in detecting conflicts between aspects.

Our final objective is to create a methodology that will improve the current state of managing conflicts in AOP and to develop tools to support that methodology.

The paper is organized in the following sections: Section 1, this section, introduced the problem of conflicts and interferences as an important issue in AOP; Section 2 will describe some proposed categorization of both aspects and conflicts; Section 3 will explain how these conflicts could be detected using unit tests; Section 4 will show how the different types of conflicts described in Section 2 can be identified by using the methodology explained in Section 3; Section 6 will introduce a small example that will help understanding how the presented methodology could be used in a real situation; Section 7 will describe several important works in the conflict detection field; finally Section 8 will list some conclusions and pointers for future work.

2. Conflicts and Interferences

Before tackling the problems posed by the introduction of aspects into an application, we have to understand the different type of changes they can perform and the objectives behind these changes.

Several attempts of categorizing aspects have been done in recent literature.

2.1 Types of Interferences

Tessier [5] classified aspects by the different type of interferences they can cause. In his work, he identified problems like:

- the use of wildcards leading to accidental joinpoints;
- conflicts between aspects and the importance of the order in which these are weaved into the application;
- circular dependencies between aspects;
- conflicts between concerns where a concern needs to change a functionality needed by another concern.

2.2 Types of Changes

Katz [6] took a different approach by classifying aspects according to the type of changes they introduce in an application. According to this author three types of aspects can be identified:

- *spectative aspects*, that only gather information about the system to which they are woven, usually by adding fields and methods, but do not influence the possible underlying computations;
- *regulatory aspects*, that change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields;
- *invasive aspects*, that change values of existing fields (but still should not invalidate desirable properties).

2.3 Types of Dependencies

Kienzle [7] approached the problem from a different point of view by considering only the dependency relationships between aspects and the original code. Three different kinds of aspect dependencies have been identified:

- *orthogonal aspects*, that provide functionality to an application that is completely independent from the other functionalities to the application;
- *uni-directional aspects*, that depend from some functionality of the application (these can be further divided as preserving, if the application functionality is maintained or enhanced without any current functionalities being altered or hidden, or modifying, if the application functionality is altered or hidden);
- *circular aspects*, which are aspects that are mutually dependent of each other.

Katz and Tessier works are extremely interesting as a starting point for our research. We intend to analyze how the different type of changes introduced by aspects (as defined by Katz) can create different types of interferences (as defined by Tessier) and how these can be tackled using unit testing. The following section explains our approach to this problem.

3. Detecting Conflicts Using Unit Tests

Unit testing is used to informally proof the correctness of modules. Each module has its own set of unit tests. By running these tests one can verify if changes to a module have changed its tested external behavior. Unit tests can be seen as a specification of the desired behavior of a module. With the introduction of aspects these specifications and their respective implementations can be easily changed by external entities, so units may no longer behave as expected. When an aspect is weaved into the code of an application, other aspects might have been weaved before and changed the expected behavior of the affected unit in a way that interferes with the new aspect being weaved.

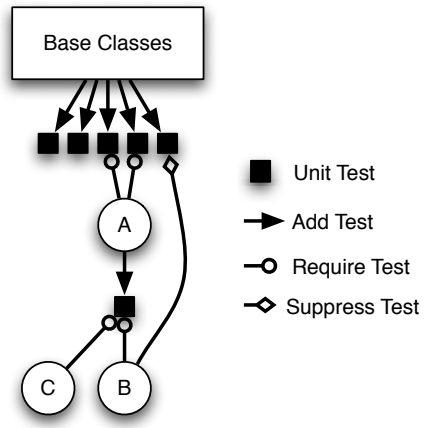

Figure 1. Aspects and Unit Tests

In this way, it should be possible to specify which unit tests have to be valid for an aspect to be correctly weaved into the system. In the same way, it should be possible for an aspect to determine which tests it expects to break.

Many times aspects depend on each other. This happens when one aspect needs some behavior to be present in the system to work properly and this behavior is introduced by another aspect. It should also be possible for aspects to introduce new unit tests into the system specifying which new behaviors are being introduced by them.

It might also happen that an aspect needs a certain behavior to be present in the system but the unit providing this behavior does not have a specific unit test for this particular behavior. Aspects should be able to add new unit tests to code already in the application.

Figure 1 shows a possible diagram for an example where aspects add, remove and depend from unit tests. In this figure the dark square boxes are unit tests. The arrowed lines identify which unit or aspect created the unit test. The circled lines identify a dependency relationship, while lines with a diamond represent an invalidation of an unit test by an aspect (the big circles). From this explanation we can see that the initial OOP code already provided several unit tests. Aspect ”A” depends on two of those unit tests and adds another one. Aspect ”B” depends on the unit test created by Aspect ”A” and at the same time suppresses one of the initial unit tests. And finally, Aspect ”C” also depends on the unit test created by Aspect A.

From this simple example we can already extract some conclusions: Aspect ”A” is probably a *spectative* aspect (as defined by Katz) that simply added some new fields and methods to the unit; Aspect ”B”, on the other hand, has probably changed the behavior of the original code. We can also easily conjecture a possible order for the weaving process (e.g. ”A” followed by ”B” followed by ”C”).

Finding a possible weaving order in which dependencies between aspects are assured can probably be accomplished by using a simple Breadth-First Search (BFS) or using the A* algorithm (as this is a typical path finding in a graph problem) . If such an ordering cannot be found then we are facing a conflict between aspects. In this case, an error message should be presented stating which aspects failed to weave, which unit tests are missing for these aspects, and which aspects removed them (if any).

The next section will explain how this approach relates to Tessier and Katz classification of aspect interferences and conflicts.

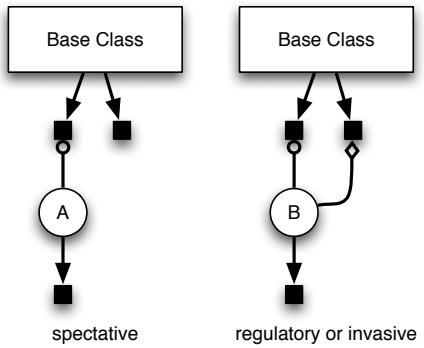

Figure 2. Different types of aspects

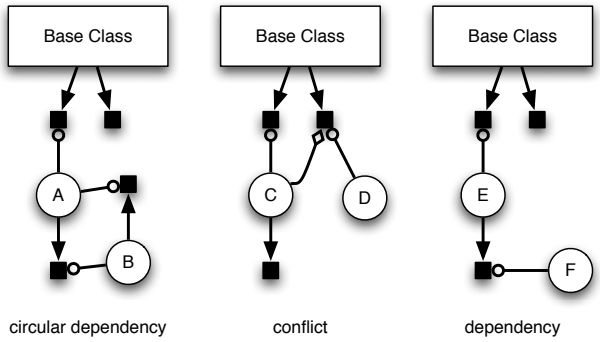

Figure 3. Different types of interferences

4. Mapping aspects and interferences to unit tests

As we have seen in Section 2, aspects can be classified as being *spectative*, *regulatory* or *invasive*. Using the notation introduced in Section 3 we can depict these different type of aspects with relation to the unit tests they add, depend on, or suppress.

In Figure 2 there are two different aspects. Aspect ”A” is probably a *spectative* aspect as it doesn't suppress any existing unit tests. It could also be an *invasive* aspect that happened to be ”lucky” enough to change something the original developer wasn't expecting to be changed and didn't include in his unit tests. Aspect ”B”, on the other hand, is clearly a *regulatory* or *invasive* aspect as it suppresses some of the original unit tests that would fail after it had been weaved into the system.

In Figure 3, three types of interferences or conflicts are depicted. In the first one, aspects ”A” and ”B” are creating a circular dependency problem. The middle diagram depicts a conflict between two concerns, where aspect ”C” is changing some functionality needed by aspect ”D”. The rightmost diagram shows aspects that need to be weaved in the correct order to function properly and at the same time a dependency between aspect ”F” and ”E”.

This shows that if unit tests are correctly used they can help detecting most of the conflicts that aspects can introduce and that have been plaguing AOP. In the following sections we will show how these conflicts can be tackled with our proposed methodology with the help of a short example a short example.

5. Using annotations to specify changes to unit tests

As has been stated before, breaking an unit test is not a clear sign of an aspect misbehaving. Due to their own nature, aspects are bound to change the functionality of other units of code and hence break their unit tests. In this way, aspects must have a way of announcing what unit tests they expect to break.

Very often, aspects are also depending on some functionality to be present into the system. This functionality can be delivered by the system base code or by other aspects. Conflicts between aspects are often caused by one aspect removing a functionality needed by another aspect, and dependency problems are commonly caused by one aspect expecting another aspect to deliver some functionality which somehow is not effectively delivered.

Therefore we claim that, there is a clear need for aspects to be able to announce which aspects they are expected to break, which aspects they depend on, and which they are adding to the overall system. In this paper we propose that aspects should be able to make this announcements using Java annotations. An example will now be introduced to explain how this could be attainable.

6. An example of conflicting aspects

Imagine a simple class depicting an User. This class would have fields like its username and password. It would also have setters and getters for those fields and a *verifyPassword* method. Listing 1 shows some simple unit tests that could have been used to ensure that the class was working properly.

Listing 1. User Class Unit Tests

```
public void testSetGetPassword() {
  user.setPassword("foo");
  assertEquals("foo", user.getPassword());
}

public void testVerifyPassword() {
  user.setPassword("foo");
  assertEquals(true, user.verifyPassword("foo"));
  assertEquals(false, user.verifyPassword("bar"));
}
```

It is also common that, for security reasons, passwords do not get stored in clear text. It is a common practice to store them using some hash function. However, to achieve a clear separation of concerns between the user data model and the security concern, this feature should be coded as a separate aspect. Listing 2 shows how this aspect could have been coded. By introducing these aspects some of the unit tests shown in Listing 1 get broken.

Listing 2. Encrypted Password Aspect

```
@SupressTest("user.UserTest.testSetGetPassword")
privileged aspect EncryptedPassword {
  protected pointcut
    passwordChanged(User user, String password):
    target(user) && args(password)
    && call(void setPassword(String));

  protected pointcut
    verifyPassword(User user, String password):
    target(user) && args(password)
      && call(boolean verifyPassword(String));

  void around(User user, String password)
    : passwordChanged(user, password)
  {
    // ... calculates md5 hash
    user.password =  md5hash;
  }
```

```
  boolean around(User user, String password)
    : verifyPassword(user, password)
  {
    // ... calculates md5 hash
    return (user.password.equals(md5hash);
  }
}
```

After introducing the aspect into the system, the developer should be warned that his aspect broke some unit tests. This could be easily computed by compiling and testing the system with and without the aspect. The developer could then inspect the broken unit test and decide if that would be an expected result from his aspect. In this case he would decide that it was because the getter and setter methods of the User class would not work as expected so he could just add a notation expressing that. The first line of Listing 2 shows how that notation could look like.

It is also common to prevent users from using passwords that are easily retrievable using brute force attacks. One way of doing it is to prevent them from using passwords that are too small. Once again, preventing this should be considered a separate aspect from the user data model and could be coded as seen in Listing 3. Notice that this aspect could have been coded in a much better fashion but for demonstration purposes it has been coded in a way that it needed the getter and setter methods of the original user class to work as originally intended. The developer should then announce that this aspect depends on the *testSetGetPassword* unit test. He could easily do so by adding a single line stating that in the beginning of the aspect.

Listing 3. Minimum Password Size Aspect

```
@RequiresTest("user.UserTest.testSetGetPassword")
@SupressesTest("user.UserTest.testVerifyPassword")
@AddsTest("user.UserTest.testVerifyPasswordML")
public aspect MinimumLengthPassword {
  protected pointcut
    setPassword(User user, String password)
    : target(user) && args(password)
      && call(void setPassword(String))
      && !within(MinimumLengthPassword);

  after(User user, String password)
  : setPassword(user, password)
  {
    if (user.getPassword().length()<6) {
      user.setPassword(password);
      throw new RuntimeException();
    }
  }
}
```

However, after introducing the aspect into the system, the developer would be warned that another aspect has suppressed that unit test. Besides that, this aspect would break the *testVerifyPassword* unit test. This is a typical case of a conflict between aspects. To solve this problem the aspect has to be rewritten in a different way and the broken unit test must be suppressed and, perhaps, a new unit test should be added to verify if everything is still working.

This example shows how unit tests, if correctly used, can help detecting conflicts between aspects. It has also shown that the developer of each different concern did not have to know about other aspects being weaved into the system, at least until conflicts occurred thus promoting obliviousness.

In the following section some of the related work done in this field will be presented and discussed.

7. Related Work

In this section some of the work that has been done in the field of conflict detection in AOP will be described briefly. The work done so far can be divided into two different categories. Automatic detection of conflicts without human intervention and forcing aspect developers to somehow express the possible points of conflict. The first two works described fall in the first category, while the remaining three fall in the second one.

7.1 Program Slicing

Balzarotti [8] claims that this problem can be solved by using a technique proposed in the early 80's called program slicing. A slice of a program is the set of statements that affect a given point in an executable program. According to the author the following holds:

> Let A1 and A2 be two aspects and S1 and S2 the corresponding backward slices obtained by using all the statements defined in A1 and A2 as slicing criteria. A1 does not interfere with A2 if $A1 \cap S2 = \emptyset$;

According to the author, this technique is accurate enough to identify all interferences introduced by an aspect but can also detect some false-positives. Furthermore, the existence of pointcuts that are defined based on dynamic contexts, forces the analysis of every execution trace increasing the number of these false-positives. However the approach has the advantage of removing the burden of having to declare formally the expected behavior of each aspect.

7.2 Introduction and Hierarchical Changes Interferences

Störzer [9] developed a technique to detect interferences caused by two different, but related, properties of AOP languages.

Störzer claims that the possibility of aspects introducing members in other classes can lead to undesired behaviors as it can result in changes of dynamic lookup if the introduced method redefines a method of a superclass. He calls this type of interference *binding interference*.

The other problem Störzer refers to is the possibility of aspects changing the inheritance hierarchy of a set of classes. He claims that this type of changes can also give place to *binding interferences* as well as some unexpected behavior caused by the fact that *instanceof* predicates will no longer give the same results as before.

To detect this kind of conflicts the author proposes an analysis based on the lookup changes introduced by aspects.

Kessler [10] also studied how structural interferences could be detected. However, his approach is based in a logic engine where programmers can specify rules (ordering, visibility, dependencies, ...). In [10], Kessler also described the different type of interferences that are possible with introductions and hierarchical changes and proposed solutions for each one of them.

7.3 Aspect Integration Contracts

Contracts have been introduced by Meyer [4] as a defensive solution against dependency problems in OOP. Some authors claim that contracts can be imported into the AOP world in order to assist programmers in avoiding interference problems.

Lagaisse [11] proposed an extension to the Design by Contract (DbC) approach by allowing aspects to define what they expect of the system and how they will change it. This will allow the detection of interferences by other aspects that were weaved before, as well as the detection of interferences by aspects that are bounded to be weaved later in the process. According to the author, for an Aspect A bound to a component C the following should be defined:

1. The aspect should specify what it requires from component C and possibly from other software components.
2. The aspect also needs to specify in which way it affects the component C and the functionality it provides (if applicable).
3. The specification of component C must express which interference is permitted from certain (types of) aspects.

This approach has the disadvantage of forcing the programmer to verbosely specify all requirements and modifications for each aspect as well as permitted interferences. On the other hand, the formal specification of behaviors has be proven to be a valuable tool in Software Engineering. However, the major drawback we se in this approach is the necessity of components to specify which interferences are permitted thus breaking obliviousness.

7.4 Regression Testing

Katz [6] 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.

Katz approach is very similar to ours but does not specify the addition and removal of unit tests by aspects. Katz argues that to use his technique one will have to specify what are the desired properties of the augmented system (after the aspect being tested is added) but does not explain how this can be done.

7.5 Service Based Approach

It has been noticed by Kienzle [7] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If some way of explicitly describing what services are requires by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

8. Conclusions and Future Work

Detecting conflicts caused by the introduction of aspects is an area where much research is being carried on and much more still needs to be done. This position paper presented a methodology that uses unit tests to tackle the this problem.

Our work, although very similar to the approaches of both Lagaisse and Katz (see Sections 7.3 and 7.4), has what we think are some major and important differences: Lagaisse forces components to specify permitted interferences thus breaking obliviousness; Katz does not specify the possibility of aspects removing unit tests in order to announce what functionalities have been changed from the original code.

Therefore, we believe our approach as promising and deserves further research and exploration to firmly confirm its value.

There is still a lot of ground to cover for this methodology to be usable. We are planning to develop plugins for some important development tools and IDEs that will allow the use of annotations to help developers on describing the behavior of their aspects. This would also allow these same development tools to give feedback about possible conflicts and interferences.

Acknowledgments

We will like to thank Miguel Pessoa Monteiro for the help in developing this paper and for the constant broadening of our perspectives. We will also like to thank FCT for the support provided through scholarship SFRH/BD/32730/2006.

References

[1] Ossher, H., Tarr, P.: Multi-dimensional separation of concerns and the Hyperspace approach. In: Software Architectures and Component Technology: The State of the Art in Research and Practice. (2000)

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: 11th Europeen Conf. Object-Oriented Programming. Volume 1241 of LNCS., Springer Verlag (1997) 220–242

[3] Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness (2000)

[4] Meyer, B.: Applying "design by contract". IEEE - Computer **25**(10) (1992) 40–51

[5] Tessier, F., Badri, M., Badri, L.: A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In: International Workshop on Aspect-Oriented Software Development. (2004)

[6] Katz, S.: Diagnosis of harmful aspects using regression verification (2004)

[7] Kienzle, J., Yu, Y., Xiong, J.: On composition and reuse of aspects. In: Software engineering Properties of Languages for Aspect Technologies. (2003)

[8] Balzarotti, D., Monga, M.: Using program slicing to analyze aspect-oriented composition (2004)

[9] Störzer, M., Krinke, J.: Interference analysis for AspectJ. In: Foundations of Aspect-Oriented Languages (FOAL). (2003)

[10] Kessler, B., Tanter, É.: Analyzing interactions of structural aspects. ECOOP Workshop on Aspects, Dependencies and Interactions (ADI) (2006)

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