

Disciplined Composition of Aspects using Tests

André Restivo
LIAAC - NIAD&R, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
arestivo@fe.up.pt

Ademar Aguiar
INESC Porto, Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
aaguiar@fe.up.pt

ABSTRACT

A large part of the software development effort is typically spent on maintenance and evolution, namely on adding new and unanticipated features. As aspect-oriented programming (AOP) can be easily used to compose software in non-planned ways, many researchers are investigating AOP as a technique that can play an important role in this particular field. However, unexpected interactions between aspects are still a major problem that compromise AOP's applicability, especially in large projects where many developers, often including new team members, are involved in the process. This paper addresses the issues of aspect conflicts and interactions and proposes a technique to help compose aspects in a disciplined way using a test-driven development approach. A simple example for a banking system helps on illustrating the technique.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Validation*; D.2.5 [Software Engineering]: Testing and Debugging—*Testing tools*

General Terms

Languages, Verification

Keywords

Aspect-Oriented Programming, Tests, Software Evolution

1. INTRODUCTION

Successful software projects do not end after their first release. As users discover bugs to be fixed and new features to be added, several code changes are required to extend and modify the existing system. Such changes can be done by the original developers, third-party developers, or even by the users themselves. Either way, changes can often lead to the introduction of more bugs and it is usually complex

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LATE Linking Aspect Technology and Evolution Workshop, April 1st, Brussels, Belgium

Copyright 2008 ACM 978-1-60558-147-7/08/04 ...\$5.00.

to understand how a new feature or a changed feature will affect the overall system.

The special characteristics of AOP languages make them good candidates for software evolution. As most evolutionary changes are unexpected, the power of AOP to change the system behavior in unforeseen ways without the need to modify the original code seems tailored for the job. However, AOP is prone to create unexpected feature interactions that can possibly lead to conflicts.

TDD is currently a popular technique for software development. Unit tests are used to test if individual units of source code work properly while integration and system tests are used to test if these same units work well together.

When using AOP, unit tests are good enough to test single units of code, but when new behavior is added by aspects, the test results might no longer be valid. In fact, new behavior might break some of the unit tests and still this might be the desired behavior for the aspects.

One possible approach would be to write code, into the aspects, that would change not only the units' behavior but also their unit tests. However, this would not suffice as other units might depend on the original behavior and tracing an integration or system test failure back to its origin might not be as easy as in classical object-oriented programming.

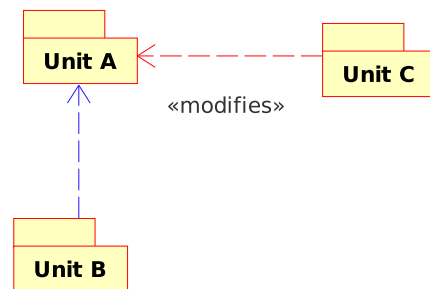


Figure 1: A simple dependency example

For example (see Figure 1), consider 2 units complemented by its unit tests that we will call *Unit A* and *Unit B*. By running its unit tests we can verify that both work correctly.

Units *A* and *B* are then integrated into a complex system and integration tests show that everything is working correctly. But after adding a new unit into the system, *Unit C*, which is a AOP unit that changes the behavior of several units in the system (including their unit tests), we find that *Unit B* stopped working.

However, *Unit C* didn't modify *Unit B*. A developer would

be puzzled by this behavior and classical TDD wouldn't help him to discover that *Unit B* stopped working because *Unit A*, which it depends on, was modified by *Unit C*.

In this paper we propose a test-based technique to compose aspect-oriented modules in a disciplined way. The technique aims at making AOP code easier to compose and more robust to evolution.

2. INTERACTIONS AND CONFLICTS

Software systems are composed of several artifacts (e.g. modules, packages, and classes) that interact with each other. Interactions occur when (i) artifacts depend upon other artifacts, (ii) modify other artifacts, or (iii) modify artifacts others depend upon.

Artifacts can be modified by other artifacts either by modifying its internal state (e.g. an object changing another object) or, when using AOP, by modifying its own behavior (e.g. an aspect changing the behavior of a class).

When two artifacts, that work independently as expected, interact in undesirable and unexpected ways, we are facing a conflict. The origin of conflicts was already thoroughly studied by several authors. Tessier [7] has identified the different types of conflicts as: accidental joinpoints, ordering problems, circular dependencies, and conflicts between different concerns (all of them unexpected and undesirable interactions).

To detect these conflicts we must first be capable of detecting interactions and then find out if they are conflicts or not. We could easily identify interactions if we knew the dependencies between artifacts, and also how an artifact is influencing other artifacts.

In our approach, we consider the relevant artifacts as the modules that compose the system. A module is a self-contained component of a system, which has a well-defined interface to the other components. In this way, a software system is composed by several modules and evolves when new modules are added or existing ones are modified.

Detecting if a module was modified by an aspect, although not trivial, was previously studied by others, of which we highlight here the work of Balzarotti [1]. However, to know a change occurred is not enough. It is also necessary to understand how the modification altered the behavior of the module and if it is harmless and desired.

The behavior of a module can be characterized by the features or services it provides. If we can find somehow that a feature was modified then we can try to understand how the behavior of the module has changed.

In the next section we will present a test-driven development approach that can help on detecting which features have changed due to the introduction of an aspect-oriented module.

3. TESTING FOR CONFLICTS

Tests are a simple way of analyzing if a feature provided by a module is correctly implemented, and was not modified by another module. Tests can be seen as incomplete specifications of the behavior of a module. If we create tests for each feature provided then we can use them to understand if a certain feature was not altered in some way.

In addition, if we also define exactly the features a module depends on, we can reason about interactions in a very simple way. However, not all interactions are conflicts. For

example, the ultimate objective of an aspect-oriented module might be to change a particular feature. Therefore, we also need to know how each module intends to change the system. This information will enable us to distinguish between desired and undesired interactions.

In our approach, a module is defined by its code, tests and also meta-information describing the features provided, modified, and all those it depends on.

By incrementally adding each module to the system we can identify if all features needed by each module are still present, and if we are modifying the system behavior in an unexpected way.

If, when a module is being added to the system, it is detected that a feature it depends on has been removed or modified by a previously added module, there is a chance that we are facing a conflict. However, modifying the behavior of a module in order to modify the behavior of modules that depend on the modified one is not uncommon. So, a module must also be able to announce that intention by deprecating dependencies and possibly adding new alternative ones.

To clarify these concepts we will introduce a small example in the next section.

4. A SMALL EXAMPLE: A BANK

Consider the example of a banking software system that stores information about customers, their accounts, and respective transactions.

One possible way to implement this system would be to develop three separate modules, one for each of these entities, with the transaction module depending on the account module, and the account module depending on the customer module.

Consider now that the software has later evolved to accommodate two new features, persistence and security, and that both would be implemented as new separate modules using AOP.

4.1 Adding Persistence

The persistence module would probably change the customer, account and transaction classes in a way that whenever a new instance of them is created or modified that would be reflected in a storage system like, for instance, a relational database.

Tests should be implemented to verify if the internal structure of each one of these classes is persistent amongst different system runs. For example, we could add a customer to an empty bank instance, recreate the bank instance and test if the customer is still present. For example, to test if the customer internal state is the same, we can serialize the customer and verify if the serialized result is the same before and after the customer has been reloaded from its persistent storage.

4.2 Adding Security

The security module would (i) add a login and password fields to the customer class, (ii) add a feature that allowed people to login into the system, and (iii) add a mechanism that would only allow new transactions to be added if the logged in person is the owner of the account in question.

Each of these features should also be tested.

The dependencies and modifications of such system are shown in Figure 2.

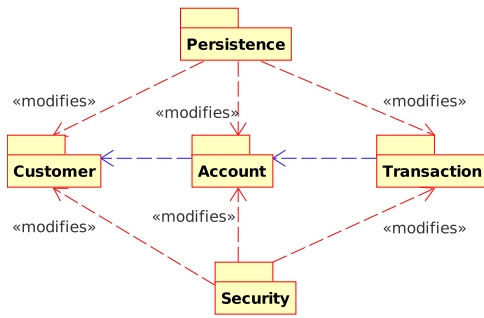


Figure 2: Module Dependencies and Modifications

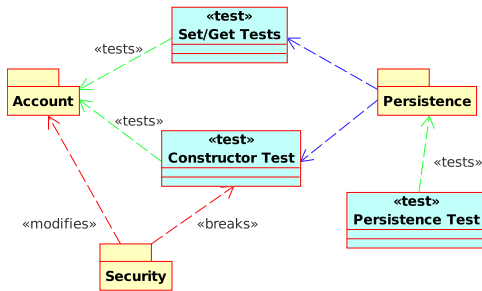


Figure 3: Security and Persistence Interactions

Both modules would announce the features they depend upon. The persistence module might depend upon the fact that the constructors, setters and getters of the base classes worked as expected.

Tests would probably exist for these features. The security module would probably need the feature that added new transactions to work properly.

Figure 3 shows the main features provided, features depended upon, and features modified by this interaction.

4.3 Persistence conflicts with Security

If the base classes are implemented correctly both modules would have the features present in the system.

However, when adding both new modules, the security module would modify the customer class in an unexpected way. If the security module is incorporated into the system after the persistence module, by running the persistence tests after the security module has been added to the system, we would identify an unexpected interaction, as the serialized version of the customer would be different from the stored one (due to the addition of the login and password fields by the security module).

On the other hand, if the persistence module is added after the security module, by running the constructor tests which the persistence module relies upon, we would discover that it was changed by the security module.

Either way, an interaction between both modules would be uncovered and it would be up to the developers to find the best way to solve it.

5. MODULE META-INFORMATION

The previous version of this technique, presented in [6], used Java annotations to store all the meta-information needed. Although sufficient for many cases, this could be insufficient

as features are seldom implemented by a single class, but rather by the collaboration of several classes. Therefore, even knowing that it is not the best solution, at the moment, we are using XML files to explicitly store the needed information. Although these files are manually written at present, they could be produced automatically by tools supporting the technique. These XML files are used to store several characteristics that we need to know about each module:

- An identifier that will allow other modules to refer to it.
- A set of features provided by the module and, for each one of them, the classes and methods used to test them.
- Features from other modules that the module depends on.
- Features the module is supposed to break intentionally.
- Dependencies from other modules that this module intentionally expects to break.

Below in Listing 1, we show an example configuration file for the security module presented in Section 4.

Listing 1 Module Definition Example

```

<module name="security">
  <feature name="accountHasOwner">
    <test class="com.bank.security.accountOwner" method="*" />
  </feature>
  <feature name="accessRestrictions">
    <test class="com.bank.security.accessTest" method="*" />
  </feature>
  <depends module="transaction" feature="addTransaction" />
</module>

```

For convenience purposes of the supporting tools developed till present, we expect that modules will be self contained in either their own directories or in jar files. This way, each module could have its own meta-information file.

The next section will explain how this technique can be used to compose a software system in a disciplined way.

6. ASPECTS COMPOSITION

The process of composing software using the technique described in this paper can be automated using the following steps:

- Find the best order in which the modules can be incrementally built into the system. This can be accomplished by using a slightly modified topological sort of the dependency graph.

Then, for each module:

- Start by verifying if all needed features are still present in the system. If they are not, warn the developer of which features are missing and which module has broken them and stop.
- Incrementally compile and add the module into the system.
- Test if all features provided by the module are working. If not warn the developer and stop.

- Test if all previously tested features are still working. If some of them fail, warn the developer and stop, unless some module states that it intentionally broke the feature.
- Test if the features that some module stated it had intentionally broken are really not working. If this is not true warn the developer and stop.
- Verify if any intentionally broken features were needed by a previously compiled module. If so warn the developer and stop.

Although a supporting tool implementing this technique was already developed, it still uses the earlier ideas presented in [6]. A new version using the XML annotation files described in Section 5 is still under development.

This tool will be capable to read the meta-information files from an entire project and incrementally compile the whole system warning the developers of any unexpected interaction it finds. The tool will also support the creation of the XML files and will provide a visual representation of the dependency graph.

Using this tool for the project presented in Section 4 is expected to be easy. After importing the AspectJ project into the tool, the user would only need to use a wizard to create each meta-information file. At first, each file would only contain information about the features provided and those it depends on. After compiling the project for the first time, the tool would warn the user about the security module changes to the customer class constructor. These changes would have been detected because a test existed for that feature or service. The user would be given the choice to add information to the XML file about that change being intentional. A second attempt at compiling the project would warn the user about the dependency between the persistence module and the customer constructor having been broken by the security module. This would allow the developer to understand exactly how both modules interact with each other.

7. RELATED WORK

Several researchers have been working to solve this same problem. In this section we present some of the other studies in this field that we consider to be the most relevant. We separated the techniques into automatic analysis techniques and user controlled analysis techniques.

7.1 Automatic Conflict Detection

Balzarotti [1] 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 [2] 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 interferences cannot be differentiated from unintentional ones.

7.2 User Controlled Conflict Detection

Lagaisse [5] proposed an extension to the Design by Contract (DbC) paradigm 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.

Katz [3] 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.

It has been noticed by Kienzle [4] 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 interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

7.3 Related Work Analysis

Automatic conflict detection techniques all suffer from the same problem. Although they are easy to use, they detect a large amount of false positives. Unless this problem can be tackled in some unforeseen way they seem to be just interesting analysis tools but not good enough for software testing and validation.

The user controlled techniques that were exposed are similar in their core ideas to our own work. Although each one uses a different approach (test, contracts and services), the main idea is always the same. Our own approach builds on these ideas by adding the notion of dependency (proposed by Kienzle) on top of Katz ideas of using regression testing.

8. CONCLUSIONS

Using AOP as a tool for software evolution is useful but due to the decoupling of features, reasoning about the code becomes harder, thus making interactions more difficult to understand. This leads to a bigger risk of conflicts and more difficulty on understanding why they happen.

This paper presented a technique based on tests that allows developers to better understand how their code is affecting other parts of the system and also to help them correct their code. Conflicts can then be detected easier and expected interactions can be described using external XML configuration files avoiding the detection of false positives.

Although recognizing that this technique does not solve all problems caused by the use of AOP, such as poorly written code or tests, or even quality code that might have conflict problems not detected by this technique, we envision that, as a complement to other analysis-based approaches, a tool based on these ideas would prove useful for software evolution using AOP.

The major difficulty that this approach yields is that it depends a lot on the developers and their skills with unit-testing. With the increasing popularity of TDD and with the help of the support tools (that are still to be developed) we believe this problem can be overcome.

9. FUTURE WORK

In the near future we plan to create a catalog containing a list of several conflict types. This catalog will also contain recipes and running examples that will allow developers to test and compare their conflict solving techniques.

We have also planned to finish and release the tool that will support the technique in question and improve the presented

method based on experiments.

The proposed approach might also be valid for other kinds of problems like using AOP in Software Product Lines or simple composition problems in AO systems. Subsequent work will explore the use of this technique in these particular cases.

Finally, we expect to be able to prove formally that the technique works for all cataloged cases.

10. ACKNOWLEDGMENTS

We will like to thank Ana Moreira, Cristina Videira Lopes, and Miguel Pessoa Monteiro for their help on constantly broadening our perspectives regarding conflicts due to aspect composition.

A word of appreciation goes also to all the reviewers that helped improve this paper with their insightful comments, namely LATE 2008 workshop reviewers.

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

11. REFERENCES

- [1] D. Balzarotti and M. Monga. Using program slicing to analyze aspect-oriented composition, 2004.
- [2] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 85–95, New York, NY, USA, 2007. ACM Press.
- [3] S. Katz. Diagnosis of harmful aspects using regression verification, 2004.
- [4] J. Kienzle, Y. Yu, and J. Xiong. On composition and reuse of aspects. In *Software engineering Properties of Languages for Aspect Technologies*, 2003.
- [5] B. Lagaisse, W. Joosen, and B. De Win. Managing semantic interference with aspect integration contracts. In *Software Engineering Properties of Languages and Aspect Technologies*, 2004.
- [6] A. Restivo and A. Aguiar. Towards detecting and solving aspect conflicts and interferences using unit tests. In *SPLAT '07: Proceedings of the 5th workshop on Engineering properties of languages and aspect technologies*, page 7, New York, NY, USA, 2007. ACM.
- [7] F. Tessier, M. Badri, and L. Badri. A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In *International Workshop on Aspect-Oriented Software Development*, 2004.