On the Existence of the AOSD-Evolution Paradox

Tom Tourwé
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
tom.tourwe@vub.ac.be

Johan Brichau
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
johan.brichau@vub.ac.be

Kris Gybels[*]
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussel - Belgium
kris.gybels@vub.ac.be

ABSTRACT

It is a well-known fact that evolving a software application accounts for the largest part of the software development process, and is currently the most problematic phase. Aspect-oriented software development is often touted as a means to ameliorate this situation: by providing new modularization mechanisms, it enables cleaner separation of concerns and reduced code tangling, and consequently makes evolving the application easier. Unfortunately, current research results only indicate that AOSD leads to applications that are better modularized, but fail to show that this improves their evolvability. Paradoxically, we have found indications of the contrary: current AOSD technologies deliver applications that are as hard, or perhaps even harder, to evolve than was the case before. We will show in this paper that the particular cause of this problem is that aspect programmers are forced to write aspects that only work for one specific version of an application.

1. INTRODUCTION

Since its conception a few years ago [11], aspect-oriented software development (AOSD) has been presented as an additional and powerful technique for improving the structure and modularity of an application. This claim is motivated by the fact that AOSD offers a new way of structuring code, besides traditional hierarchical decomposition, by means of so-called aspects. These aspects bundle *crosscutting* code, e.g. code that would otherwise have been spread throughout the whole application. At the same time, it is widely acknowledged that the modularity of an application has a significant influence on its evolvability. Intuitively, the more modular an application, the easier it is to evolve. Given the fact that evolution of software applications accounts for the largest part of the software development process, the introduction and use of AOSD techniques looks promising.

[*]Johan Brichau and Kris Gybels are research assistants of the Fund for Scientific Research, Flanders, Belgium (F.W.O.)

Unfortunately, current research in the AOSD community mainly focusses on application development only, and much less on the long-term behavior of applications developed with aspect technology. In other words, much attention is paid to developing highly modular applications, but there seems to be much less interest in studying the effect(s) of AOSD on evolution. This observation is quite strange, given that evolution consumes most of the development time of an application.

Paradoxically, we've found that current AOP technologies deliver programs that are as hard or even harder to evolve than was the case before. In our view, this is due to the following two problems:

1. Current industrial software development environments, such as [8, 3], offer sophisticated means for evolving an application, often by means of refactoring techniques [12, 5]. While AOSD techniques are currently being incorporated into such environments (e.g. Eclipse [8]), their impact on the support for evolution offered by these environment is largely neglected. As such, the developer has no support for evolving the aspects. For example, the current support for evolving a base program may overlook code that is included in an aspect. This will most likely result in an application that is inconsistent and does not exhibit the desired behavior.
2. AOSD technology puts forward the assumption that the base program is syntactically oblivious to the specific aspects that are applied upon it [4]. The aspects themselves, on the other hand, have to include a crosscut description of all places in the application where this code yields an influence (e.g. pointcuts in AspectJ). Consequently, it is much harder to make such crosscuts oblivious to the application. Moreover, current means for specifying crosscuts rely heavily on the existing structure of an application. As such, the aspects are tightly coupled to the application, and it is a well-known fact that tight coupling can seriously hamper the evolvability of an application [13]. In this particular case, it is clear that when the application evolves and its structure changes, all crosscuts in every aspect may have to be checked, and possibly revised.

The first issue can be solved by studying how refactoring, or any other form of supporting evolution at a high conceptual level, can include support for aspects as well as for reorganizing code that is included in an aspect. This is not the focus of this paper. The second issue, however, is exactly what we want to explain, since it is more fundamental, much harder to solve, and leads to what we call the *AOSD-Evolution paradox*: the observation that AOSD leads to software that should be more robust with respect to evolution because it offers better modularization, but paradoxically reduces the evolvability because it introduces tight coupling. In the remainder of this paper, we will further investigate this issue, by providing some explanatory examples, identifying the major causes for this phenomenon, and providing hints for possible solutions.

2. PROBLEM STATEMENT

In this section we will show why current AOSD technologies inevitably lead to programs that are not easier to evolve. As a running example throughout the section, we will consider an abstract framework that we will present first. In the discussion that follows, we will distinguish between two different kinds of developers: a *framework developer*, who is responsible for developing the initial version of the framework, and a *framework maintainer* who is responsible for maintaining the framework and changing it as required. Of course, in practice, one single developer may be a framework developer and a framework maintainer at the same time.

2.1 Introduction: An Example Object-Oriented Framework

The problem we want to show particularly manifests itself in the context of object-oriented frameworks [9, 10]. A framework is a skeleton application, offering a flexible design intended to be reused when building an application within the particular domain of the framework. To this extent, the framework defines so-called *hot spots* [14], that identify those places in the design where application-specific code can be added. Typically, a framework defines a number of constraints upon these hot spots, that should be adhered to at all times by the application-specific code in order to arrive at an application exhibiting the desired behavior. Such constraints range from simple naming conventions, over specific coding patterns that should be followed, to complex object interactions that the code should provide [1, 6].

Consider the abstract example of an object-oriented framework depicted in Figure 1. It consists of a simple class hierarchy, `A`, that includes a number of subclasses and defines a method `m`. This method is overridden in each of the various subclasses of the hierarchy. This is an example of a naming convention that is defined by the framework: the signature of the `m` method should of course be the same for each and every subclass. For the sake of the discussion, the example abstracts away from other specific constraints and coding conventions.

2.2 Requirements

From the point of view of a framework maintainer, the principle of separation of concerns simply means that he should not be forced to consider every individual concern when implementing another one. For example, when adding new functionality to some classes in an application, the maintainer should not be obliged to deal with a *synchronization* concern. This is the exact reason why AOSD technologies allow developers to cleanly separate different concerns and provide additional composition mechanisms to weave these concerns into a working application.

As we will discuss in the next section, however, it appears as if current AOSD technologies force a framework developer to write crosscuts in such a way that a framework maintainer will need to remember to change these when he changes the framework.

2.3 The Problem

A framework developer is responsible for implementing a framework that incorporates the desired functionality. Inevitably, the framework will contain several crosscutting concerns that can be implemented using AOSD technology in order to improve modularity. As we will show, it is however very difficult to capture the required joinpoints of the aspects in a general and expressive manner, due to the simplistic crosscut languages used today by current AOSD languages. As a consequence, the framework developer will find himself forced to writing suboptimal crosscuts which break his aspects at the slightest change applied to the framework itself.

2.3.1 Implementing the Aspects

Consider the situation where a framework developer wants to define an aspect that should influence only some methods `m` defined in subclasses of the class `A`, but not all methods `m`. To implement the corresponding crosscut, he could simply write one explicitly enumerating these methods using the name of the class and the method itself. Obviously, this creates a burden for the framework maintainer. Whenever a new implementation for the method `m` is added (in some new class or an already existing one that does not yet define `m`) that should be influenced by the aspect, the enumeration needs to be changed. To avoid this situation, the framework developer would rather like to uncover a common pattern in the methods that need to be captured by the crosscut. Then, he can encode this pattern so that new methods `m` that need to be influenced by the aspect are captured automatically. This can turn out to be more difficult than it seems however.

For starters, the common pattern can only be found in the implementation of the methods. Because of the framework's design, all methods should be named `m`, those that should as well as those that shouldn't be influenced, are all the same, as are the types of their arguments and return values. The only differing element is the class of the methods, but as discussed above, using that would only lead to an undesirable enumeration.

Given the current techniques available in AOSD tools, considering method implementations inside a crosscut can only be achieved by using common coding conventions or patterns used by those methods. For example, in AspectJ, which has one of the most advanced crosscut languages, there is no way of doing so. In AspectJ, a developer can specify crosscuts in terms of the dynamic behavior of the application. But nevertheless, the `cflow` and `if` constructs are the only primitive pointcut expressions that allow to describe a pointcut beyond the 'local' scope.

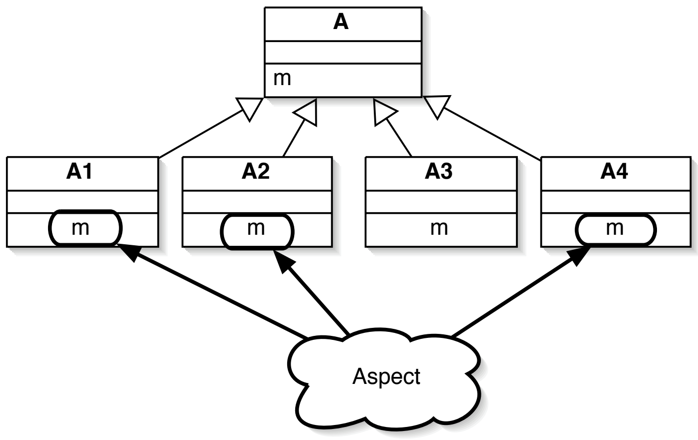

Figure 1: An Example Object-Oriented Framework

This is far from sufficient to separate out those methods that the crosscut should capture. For one, it cannot be assured that all methods to be included are in the control flow of a particular other joinpoint. Second, since the `m` methods actually form a hot spot of the framework, they each carefully follow the necessary coding conventions and obey the appropriate constraints. Therefore, it cannot be guaranteed that only the methods to be included in the crosscut each follow a particular pattern at all. Other `m` methods may do so as well.

In other words, the problem occurs because current AOSD techniques try to discriminate methods based on some common structural property (which can be a combination of other, more simple properties), such as particular naming conventions, coding conventions and coding patterns. In the context of large-scale industrial frameworks, this approach suffers from the following problems:

- The framework itself also defines such conventions, and classes and methods should adhere to these in order to implement the correct behavior. The conventions defined by the framework may conflict with those used for defining crosscuts.
- It can not be guaranteed that the methods that should be captured by a crosscut all share this property. Neither can it be assured that no other methods share this property.
- it is a well-known fact that developers do not always adhere to particular coding conventions or patterns [16]. This has several reasons: the complexity and size of the application, the constant evolution of the application, the lack of sufficient and up-to-date documentation, and the lack of active support for automatically checking such conventions [15].

Consequently, there are situations in which a developer has no choice but to use simple enumeration of the various joinpoints to make up a crosscut.

2.3.2 *Refactoring: A Naive Solution*

One argument that is often raised in situations such as the one sketched above, is that the application could be refactored in order to ensure that the intended methods can be captured more easily. Such an example of refactoring for the sake of aspects can be found in [2]. As far as we are concerned, this is not the case. Most importantly, we believe an application should not be refactored with the sole intent of enabling a developer to define an aspect more easily. Refactoring should only be applied when the current structure of the application *smells bad* [5], and its quality should be restored. There are other problems with refactoring as well, however.

Consider the example in Figure 1, it is clear that we can never refactor the class hierarchy so that we can use naming conventions to identify the correct joinpoints. The hierarchy constrains the implementation and requires that all `m` methods share the same signature. The only other option consists of reorganizing the different `m` methods so that it becomes possible to use coding conventions to define the crosscuts. The idea would be to separate the methods into those that share some common structural property, and those that do not adhere to this same property. We believe such an approach to be impractical, at the least, and impossible at the worst, in most situations, however.

First of all, suppose it would be possible to factor out one common property between all methods that should be captured in a particular join point. This can be achieved by splitting the implementation of the methods in multiple auxiliary methods, for example, or by inserting an intermediate superclass in the hierarchy, that captures the common property. In both cases, however, the need for using an aspect disappears. The specific purpose of an aspect is exactly to capture a crosscutting concern, that can not be factored out by using traditional decomposition techniques.

Second, a realistic application contains a multitude of aspects, which increases the likelihood that one single method is captured by multiple aspects. Consider as an example the framework in Figure 2, on which three different aspects

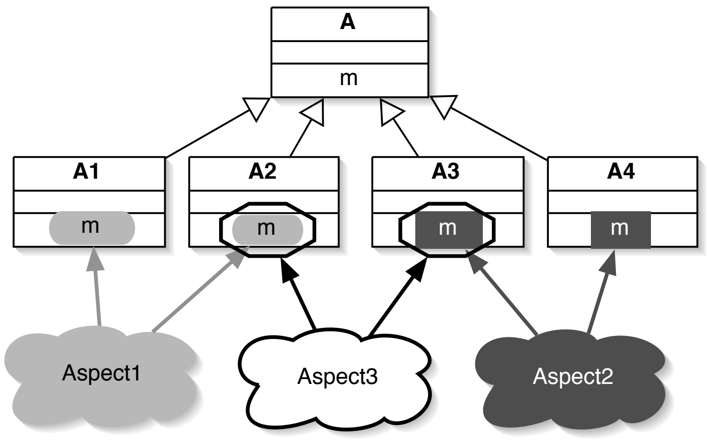

Figure 2: An Example Object-Oriented Framework with Multiple Aspects

are applied. Each aspect captures various different implementations of the method m in its crosscut definition. Using present AOSD technologies, a developer must manage to find a set of discriminating properties of all methods to describe the different crosscuts. To accomplish this, the crosscut of each aspect should be based upon a particular coding convention. However, methods participate in more than one aspect, and they can not easily obey many different coding conventions at the same time. Reorganizing the methods for either aspect might thus interfere with a reorganization of the same methods for any other aspect.

Clearly, any reorganization of the existing structure of an application so as to ensure that aspects can be defined more easily is very difficult, if not impossible. Given this observation together with those made in the previous section, the only conclusion that can be drawn is that under some circumstances, developers have no choice but to resort to enumeration-based crosscuts.

3. SUMMARY & DISCUSSION

3.1 Summary

We strongly believe the problem we have sketched above manifests itself because all major AOSD techniques in use today provide very limited means to specify crosscuts. Basically, developers have no choice but to specify such crosscuts based on structural properties, such as naming or coding conventions. As we have seen, this creates three kinds of problems, in order of increasing importance:

- It is a well-known fact that agreed upon conventions are more often violated then they are adhered to in large-scale and complex applications. This has several causes, such as incomplete or inconsistent documentation or deadline pressure, which are not easily solved.
- We feel that an application should not be refactored, with the sole intent of enabling a developer to define an aspect more easily. An application should be oblivious as to which aspects are applied upon it.
- In the context of object-oriented frameworks that contain many different aspects, it is often impossible to use such conventions for expressing crosscuts, since these conventions conflict with the conventions and constraints imposed upon the application by the frameworks, or even with the conventions used by other aspects.

3.2 Towards a Solution?

The one and only way of tackling this problem is using a more sophisticated and expressive crosscut language. This would allow a developer to define crosscuts in a more intentional way, and would enable him to discriminate between methods based on what they actually do instead of what they look like. It is still to be determined however if developers can describe aspects that can be applied to any version of the framework without revisions to the crosscuts. Crosscut languages are thus clearly an important topic for research in AOSD and it seems that this issue is largely neglected.

In [7], we describe how the characteristics of a logic metalanguage enhance the description of crosscuts. In that paper, we mainly described how particular language features are beneficial for writing more intentional crosscut expressions. The approach can actually be seen as a derivative of the AspectJ crosscut language because it is mostly based on the same join point model as AspectJ. The most important difference is that we use full predicate logic to describe the crosscuts and provide the crosscut programmer with a complete set of predicates to investigate the entire lexical structure of the program. Unfortunately, the most important research question remains, i.e.: what should be the meta representation of the base program such that intentional crosscuts can be written in terms of it. Since the conception of AOSD, we have seen full reflective models, static and dynamic join point models, event-based models, etc.... Most of these models are either 'ad hoc' or are driven by the requests of users. Now that we have seen the first requirements and uses for AOSD technologies, we think the time is ripe for an extensive study in the language design for crosscut languages.

4. CONCLUSION

In this paper, we have shown that aspect technology, although claiming to improve the evolvability of an application, actually introduces a number of serious issues that degrade this evolvability at the same time. We termed this phenomenon the *AOSD-Evolution* paradox. We have shown how simple crosscut languages, such as those used in all current major AOSD incarnations, often force a developer to resort to enumeration-based crosscut expressions, which clearly has a negative influence on the evolvability of the application. We argumented that, to alleviate this problem, more powerful and expressive crosscut languages are needed, that allow a developer to write down a crosscut in a more intentional way. Only in that case will it be possible to discriminate between methods that define different semantics but share similar structural properties. Consequently, it would be a first and important step towards making aspects more oblivious to the application upon which they are applied.

5. REFERENCES

[1] K. Beck. *Smalltalk Best Practice Patterns.* Prentice Hall, 1996.

[2] Y. Coady and G. Kiczales. Exploring an Aspect-Oriented approach to OS code. In *Workshop on Advanced Separation of Concerns at OOPSLA 2000.*

[3] I. Corporation. Intellij idea, http://www.intellij.com/idea.

[4] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000,*. 2000.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[7] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *To be published in Proceedings of AOSD 2003*, 2003.

[8] O. T. International. The Eclipse Platform, http://www.eclipse.org.

[9] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1988.

[10] R. E. Johnson and V. F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, June 1997.

[12] W. F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana Champaign, 1992.

[13] Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[14] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley/ACM Press, 1995.

[15] T. Tourwé. *Automated Support for Framework-Based Software Evolution.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[16] J. van Gurp and J. Bosch. Design Erosion: Problems & Causes. *Journal of Systems & Software*, 61(2):105–119, 2001.