Automatic Support for the Migration Towards Aspects

Mariano Ceccato
Fondazione Bruno Kessler—IRST, Trento, Italy
ceccato@fbk.eu

Abstract

Aspect Oriented Programming (AOP) has been proposed as a new programming paradigm. The originality in AOP is the aspect, a single modularization unit for all those functionalities that were originally spread across several modules and tangled with each other (called crosscutting concerns). Using an aspect, a crosscutting concern can be factored out into a single, separate unit.

This paper summarizes a PhD thesis that presents an approach to automatize the migration of existing Object Oriented systems towards AOP. Different techniques are proposed to cope with the migration and assessed on a large software basis.

1 Introduction

One of the main drawbacks in adopting any modularization strategy in software development is that there are system functionalities that can not be assigned to a single module in the chosen decomposition. Examples of functionalities that suffer this problem are persistence, error management and logging. Since the code fragments that implement these concerns are spread across many units, they are called *Crosscutting Concerns*.

Crosscutting concerns violate the modularization goal that a system is decomposed into small independent parts. Crosscutting concerns are transversal with respect to the units in the principal decomposition, because their implementation consists of a set of code fragments distributed over a number of units. The maintenance of such a scattered concern could pose understandability problems.

Aspect Oriented Programming [7] (AOP) has been proposed to solve the main problems of crosscutting concerns (namely scattering and tangling), by providing a unique place where the related functionalities are implemented. A new modularization unit, called *aspect*, can be defined to factor out all code fragments related to a common functionality, otherwise spread all over the system. For example, an application can be developed according to its main logical decomposition, while the possibility to serialize and deserialize some of its objects can be defined in a separate aspect.

In order to extend the benefits of AOP to already existing systems, a significant reverse and re–engineering effort is required. The effort consists, first of all, in analysing the existing application source code looking for those portions that implement the crosscutting functionality. The second part of the work is the transformation of the existing program into an aspect-oriented reformulation.

This paper presents a summary of the PhD thesis [1, 2]. Aspect mining and refactoring are presented in Section 2 and Section 3. Then, the assessment of the whole process is summarized in Section 4 and, eventually, Section 5 closes the paper.

2 Aspect Mining

2.1 Aspectizable interfaces

When a program is designed according to the OOP paradigm, the hierarchy of the classes reflects the (principal) decomposition of data structures and functions into smaller, composable units. In such a decomposition, the interfaces play a twofold role:

1. An interface may collect abstract properties of the principal decomposition, shared by the classes implementing it.

2. An interface may collect transversal properties, that crosscut the principal decomposition. Such properties recur across multiple unrelated classes, instead of being confined within a single, cohesive group of classes.

We call the latter an *aspectizable interface*.

Let us consider persistence (e.g., interface `Serializable` in the Java standard library). The code fragments implementing this interface are spread across several classes (scattering). Moreover, this code requires access to information about each entity to be stored persistently (tangling). On the other side, if we consider the persistence functionality from a logical point of view, it clearly does not belong to the principal decomposition of the application. Rather, it is a transversal computation that has to be superimposed. In other words, it is an *aspect* of this application.

We have defined the following set of aspect mining indicators specifically for the migration of interface implementations to aspects. The implementation of an interface is marked as a candidate aspect when:

- **External package:** The interface implemented in a class belongs to a package different from that of the given class.
- **String matching:** The name of the interface implemented in a class matches a user defined pattern (e.g., `".*able"`).
- **Clustering:** When methods are clustered according to the call relationship, interface methods are not grouped together with other (non-interface) class methods.
- **Unpluggability:** The methods of the interface implemented in a class can be unplugged from the given class, since they are not invoked by other methods of the same class.

2.2 Dynamic aspect mining

We propose to use feature location [3] for aspect mining according to the following procedure. Execution traces are obtained by running an instrumented version of the program under analysis for a set of scenarios (use cases). The relationship between execution traces and executed computational units is subjected to formal concept analysis (FCA [5]). The execution traces associated with the use cases are the *objects*[1] of the concept analysis context, while the executed class methods are the *attributes*.

Based on the resulting concept lattice (with sparse labeling), the notion of *use-case specific* concepts and *generic* concepts are defined.

Both use-case specific concepts and generic concepts carry information potentially useful for aspect mining, since they group specific methods that are always executed under the same scenarios. When the methods that label one such concept crosscut the principal decomposition, a candidate aspect is determined.

A concept is a reported as crosscutting concern if these two symptoms are identified:

- *scattering:* more than one class contributes to the functionality associated with the given concept;
- *tangling:* the class itself addresses more than one concern.

[1]Not to be confused with objects in Object-Oriented programming.

2.3 Mining combination

Dynamic aspect mining revealed some limitations. A limitation is the ability to report only portions (*seeds*) of actual crosscutting concerns, due to the fact that dynamic aspect mining relies on a limited set of execution scenarios. In order to overcome its limitations, we combine dynamic aspect mining with relevant mining techniques taken from the literature, they are fan-in analysis [8] and identifier analysis [10].

Fan-in analysis is based on the assumption that a method called from many different places (i.e., with a high fan-in) represents a seed for a crosscutting concern, in that the call sites are spread throughout the system. Only methods whose fan-in is above a given threshold are reported as potential crosscutting concerns.

Fan-in analysis and dynamic aspect mining are quite complementary. In fact, while fan-in analysis focuses on identifying those methods that are called at multiple places, dynamic aspect mining discards them, because such methods are likely to occur in many execution traces. A method with a high fan-in is not specific to any use-case. A combination of fan-in analysis and dynamic aspect mining consists in applying each technique individually and taking the union of the results.

In identifier analysis [10], naming convention is used to identify crosscutting concerns. Identifiers are split into words according to the contained capital characters (camel casing). For instance, the class name *QuotedCodeConstant* generates the substrings *quoted*, *code* and *constant*. Formal Concept Analysis is then applied to this word list to group together source code entities, when they share similar names. Groups correspond to the aspectual views that should guide developers in the crosscutting concern identification.

Identifier analysis produces very interesting results, because it focuses on complete concerns and not just on seeds. However, many false positives are reported and a quite time-consuming manual intervention is required to filter them out. Consequently, better results could be achieved if identifier analysis is used as seed expansion technique for the seeds identified either by fan-in analysis or by dynamic aspect mining (or by their combination). In this way, the search space for identifier analysis can be significantly reduced and less false positives are expected.

Considering respective strengths and limitations, we propose to combine the mentioned techniques according to the subsequent algorithm:

1. Identify interesting candidate seeds by applying fan-in analysis, dynamic aspect mining or both;
2. For each method in the candidate seed, find its enclosing class, and compute the identifiers occurring in the

method and the class name, according to camel casing;

3. Apply identifier analysis to the application, and search for the concept, among the concepts it reports, that is "nearest". The nearest concept is the concept that contains most of the identifiers generated in the previous step. If more than one nearest concept exists, take the union of all their elements.

4. Add the methods contained in the nearest concept(s) to the candidate seed (seed completion).

5. Revise the expanded list of candidate seeds manually to remove false positives and add missing seeds (false negatives).

3 Refactoring

3.1 Aspectization of the aspectizable interfaces

The refactoring of an aspectizable interface consists of modifying all the classes that realize such interface, where all the interface methods are moved from the classes to an aspect. Migration of an aspectizable interface to an aspect involves two main, high-level code transformations (refactorings, [4]):

1. *Move properties to aspect*: properties (attributes, methods, inner classes) are modularized in the aspect, that introduces them into the affected classes.

2. *Remove references to properties*: execution points referencing aspectized properties are moved into the aspect code (called *advice* code) triggered by pointcuts.

In the case of the aspectizable interfaces, the first transformation is the most important one, since the methods in the interface implementations are seldom referenced by methods in the principal decomposition.

The overall transformation can be described in terms of three simpler refactoring steps, applied repeatedly. They are:

- *Move method to aspect.*

- *Move field to aspect.*

- *Move inner class to aspect.*

These three (atomic) refactorings consist of removing a method (resp. field or inner class) from a given class and adding it to an aspect, where it becomes an introduction.

3.2 Pointcut extraction

The identification of aspect candidates (aspect mining) and the mark-up of the code regions that represent instances of these candidates to be refactored are assumed to have been completed before the extraction begins.

The pointcut extraction process consists of a loop over all the marked portions of code. In turn, each marked segment of code is analyzed in isolation and, with the involvement of the user, moved to an aspect. The iteration goes on until no more marked statements remain in the base code.

Among the high number of refactorings for migration from objects to aspects that have been proposed (e.g., in [6, 9, 11]), we focuses on a small subset, selected using a mix of a bottom-up and of a top-down approach. A-priori knowledge about the constructs provided by most aspect languages/frameworks (including AspectJ) and about the kinds of crosscutting concerns that are amenable for implementation through aspects guided us. Furthermore, we considered only refactorings whose mechanics can be fully automated. In fact, one of our objectives is to assess the ability of this small list of refactorings to cover most of the cases encountered in practice. The following six refactorings from objects to aspects have been included:

- **Extract Beginning/End of Method/Handler:** The marked code is at the beginning/end of the enclosing method body or of one of the method's exception handling blocks.

- **Extract Before/After Call:** The marked code is always before or after a method call.

- **Extract Conditional:** A conditional statement controls the execution of the marked code.

- **Pre Return:** The marked code is just before the return statement.

- **Extract Wrapper:** The marked code is part of a wrapper pattern, in which the wrapper code is to be aspectized.

- **Extract Exception Handling:** The marked code is a whole exception handling block.

When none of the refactorings above apply to a marked code fragment, OO transformation is resorted to (also called OO refactorings [4]) in order to make one or more of the refactorings above applicable. Among the possible OO transformations, the following are regarded as the two most important ones: *Statement Reordering* and *Extract Method.* Both can be fully automated (the latter is available in most transformation environments, while the former requires non trivial dependency analysis to ensure semantics preservation).

Program	Size (LoC)	Techniques
JDK classes	382,533	Aspectizable interfaces
JHotDraw	40,022	Dynamic aspect mining, Mining combination, Aspectizable interfaces, Pointcut extraction
FreeTTS	31,009	Aspectizable interfaces
JGraph	18,373	Aspectizable interfaces
PetStore	17,032	Pointcut extraction
JSpider	13,979	Pointcut extraction
JAccounting	11,676	Pointcut extraction
Total	**514,624**	

Table 1. Java programs used as case studies.

4 Assessment

All the proposed approaches for aspect mining and aspect refactoring have been applied to existing software systems. Source code used in the assessment comes from selected open source Java projects, accounting for more that half a million lines of code (see Table 1). They range from component oriented classes to full applications with intermediate cases.

The migration toward AOP is expected to be beneficial for external quality attributes such as understandability and maintainability, as well as for internal quality attributes, such as modularity. In fact, the possibility to encapsulate separate concerns should result in a localized comprehension and modification effort, and an improved code structure. So, we designed an empirical study (controlled experiment) to measure how these qualities improve during the migration. We asked some software developers to work in a controlled environment and perform selected maintenance tasks either on an OOP system or on its AOP version (after aspectizable interfaces extraction).

For reason of space, we do not report here data about the assessment of mining and refactoring. More details can be found in the thesis [1]. Instead, we briefly summarize here our findings about the controlled experiment.

Overall, the study indicates that the migration of the aspectizable interfaces has a limited impact on the principal decomposition size, but at the same time it produces an improvement of the code modularity. From the point of view of the external quality attributes, modularization of the implementation of the crosscutting interfaces clearly simplifies the comprehension of the source code. We hypothesize that further benefits in the overall maintainability would be achieved if a larger fraction of the code was affected by the migration to AOP. However, further experiments are necessary to validate this hypothesis.

5 Conclusion

This paper summarizes the main contribution of the PhD thesis on the migration of existing source code toward AOP. Several techniques have been proposed and assessed both for the identification of candidate aspects and for their transformation into actual aspect code. All the proposed techniques have been implemented into prototypes and applied to some case study software systems. Moreover, a controlled experiment has been conducted to evaluate the benefits achieved after the migration in terms of source code understandability and maintainability.

References

[1] M. Ceccato. *Migrating Object Oriented code to Aspect Oriented Programming*. PhD thesis, University of Trento, Italy, December 2006.

[2] M. Ceccato. Migrating object oriented code to aspect oriented programming. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance, 2007 (ICSM 2007)*, pages 497–498. IEEE, Computer Society, October 2007.

[3] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[4] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[5] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[6] J. Hannemann, G. C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM Press.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proc. of the 11th European Conference on Object Oriented Programming (ECOOP), vol. 1241 of LNCS*, pages 220–242. Springer-Verlag, 1997.

[8] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. of the 11th IEEE Working Conference on Reverse Engineering (WCRE 2004)*, pages 132–141, Delft, The Netherlands, November 2004. IEEE Computer Society.

[9] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122. ACM Press, March 2005.

[10] T. Tourwe and K. Mens. Mining aspectual views using formal concept analysis. In *Proc. of SCAM 2004*, pages 97–106, Chicago, Illinois, USA. IEEE Computer Society.

[11] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*, Waterloo, Canada, November 2003.