Migrating Object Oriented code to Aspect Oriented Programming

Mariano Ceccato
Fondazione Bruno Kessler—IRST, Trento, Italy
ceccato@itc.it

Abstract

Aspect Oriented Programming (AOP) is a new programming paradigm that offers a novel modularization unit for the crosscutting concerns. Functionalities originally spread across several modules and tangled with each other can be factored out into a single, separate unit, called an aspect.

We investigated automated techniques that can be used to support the migration of existing Object Oriented Programming (OOP) code to AOP. To migrate an application to the new paradigm, a preliminary identification of the crosscutting concerns is required (aspect mining). Then refactoring is applied to transform the scattered concerns into aspects. The proposed methods have been assessed on case studies for a total of more than half a million lines of code.

1 Motivation

Software systems are so complex that they can not be developed without dividing them into sub-modules. The main drawback in this approach is that there are some system functionalities that can not be assigned to a single module in the system 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: their main characteristic is that they are transversal with respect to the units in the principal decomposition, i.e., their implementation consists of a set of code fragments distributed over a number of units.

Reasoning about crosscutting concerns can be quite difficult, because it requires to deal with a lot of modules at the same time, in that there is no modularization support for them. For example, to deal with the persistence functionality, we have to understand all the pieces of code that perform persistent storage and retrieval. Crosscutting concerns are sources of problems, because their modification requires that all code portions where such a functionality is implemented be located (problem: scattering) and that all ripple effects associated with the changes be determined (problem: tangling).

Aspect Oriented Programming [5] (AOP) aims at solving the two 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 de-serialize some of its objects can be defined in a separate aspect.

2 Problem definition

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, of 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.

In this work, the problem of migrating an existing system from OOP to AOP has been addressed by dividing it into its two composing phases. Identification of the code portions that are most suitable for migration to aspects is conducted during the aspect mining phase, in which the source code is analyzed and candidate aspects are located. Then, in the refactoring phase, the code is transformed, so that crosscutting concerns are realized by separate aspects instead of the original classes. The theoretical properties of the proposed methods are validated by a number of experiments.

During migration, human guidance is both necessary and desirable; the process requires value-judgments regarding trade-offs best made by a maintenance engineer. The process of migrating existing software to AOP is highly knowledge-intensive and any migration toolkit therefore should include the user in the change-refine loop. However, notwithstanding this inherent human involvement, there is

considerable room for automation.

3 Contribution

The contribution consists in the definition and in the assessment of a migration process for existing software systems from traditional programming paradigms (OOP) to aspect oriented programming. This contribution is articulated into many results [3].

We defined a novel method to identify automatically the crosscutting concerns present in an existing OOP application, *dynamic aspect mining* [7]. It is based on the analysis of the traces of use-case executions. The only assumption is that some of them exercise the crosscutting functionalities to be separated into aspects. This corresponds to traceability from requirements (that correspond to use-cases) to implementation. In fact, whenever a requirement has a scattered and tangled implementation, it is possible to define a use-case for such a requirement, which exercise precisely the non-modularized functionality.

The same technique has been compared with two other approaches (fan-in analysis [6] and identifier analysis [9]), by applying all of them to a common benchmark case [4]. They have been mutually compared and their respective strengths and weaknesses have been assessed. Moreover, by identifying where techniques overlap and where they are complementary, interesting combinations have been proposed.These combinations have been applied to the same benchmark application to verify whether they perform better than the original techniques.

We defined the notion of aspectizable interfaces [8] as those interfaces that collect transversal properties that crosscut the principal decomposition, in contrast to the more standard notion of interface which collects abstract properties of the principal decomposition, shared by all the classes implementing such an interface.

A technique was then proposed for the aspectization of interface implementations. Identification of the interfaces that are most suitable for migration to aspects is conducted during an aspect mining phase, in which the source code is analyzed and candidate aspects are located. Then, in the refactoring phase, the code is transformed, so that interface implementations are realized by separate aspects instead of the original classes. We have implemented a toolkit to support the aspectization of interface implementations and we have applied it to the source code of some existing applications. The aim of the experimental work was to assess the feasibility of the transformation and to evaluate the potential benefits.

Six refactorings have been introduced to support migrationfrom OOP to AOP. They have been combined with existing OO transformations in a tool that automates them [1]. The effectiveness of the approach was investigated and some case studies provide evidence suggesting that migration can be achieved with a simple set of refactoring transformations [2]. The case studies also point to the importance of enabling transformations which transform an OO program into a semantically equivalent OO program, in which the refactorings become applicable.

All the proposed techniques have been implemented into tool prototypes which are publicly available. Such tools have been used to apply the presented methods on a large set of case study applications for a total of more than half a million lines of code. In particular, one of these applications, JHotDraw[1], has been subjected to all the methods. In order to measure the effects of migration,an empirical study has been conducted. It allowed us to assess the benefits achieved after migration, in terms of source code understandability and maintainability [8].

References

[1] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *Proc. of the International Conference on Software Maintenance (ICSM 2005)*, pages 27–36. IEEE Computer Society, September 2005.

[2] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.

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

[4] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, September 2006.

[5] 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.

[6] 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.

[7] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working conference on Reverse Engineering (WCRE 2004)*, pages 112–121. IEEE Computer Society, November 2004.

[8] P. Tonella and M. Ceccato. Refactoring the aspectizable interfaces: an empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, October 2005.

[9] 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.

[1] http://www.jhotdraw.org/