Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 51 (2009) 138–149

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Object-oriented transformations for extracting aspects

Marcelo Nassau Malta, Marco Tulio de Oliveira Valente *

Institute of Informatics, PUC Minas, Belo Horizonte, Brazil

Received 24 October 2007; received in revised form 25 January 2008; accepted 7 February 2008
Available online 15 February 2008

Abstract

In the migration of object-oriented systems towards the aspect technology, after locating fragments of code presenting a crosscutting behavior and before extracting such code to aspects, transformations may be needed in the base program. Such transformations aim to associate crosscutting code to points of the base program that can be captured using the pointcut descriptor model of aspect-oriented languages. In this paper, we present a catalog of object-oriented transformations and demonstrate the importance of such transformations by reporting on a case study involving four systems that have been aspectized using AspectJ.
© 2008 Elsevier B.V. All rights reserved.

Keywords: Aspect-oriented programming; Refactoring; Software evolution; Program transformation

1. Introduction

In the last 10 years, aspects have emerged as the main programming paradigm for the modularization of crosscutting concerns. As a consequence, several works have been developed in two areas that are critical when migrating to this new technology: aspect mining [6,19,34] and aspect-oriented refactoring [21,16,24,25]. The purpose of aspect mining techniques is to identify crosscutting concerns in legacy, non-aspect-oriented code. Once they are located in the target program, aspect-oriented refactorings can be applied to modularize such concerns into equivalent aspects. As usual in refactoring tasks, the purpose is to improve the internal structure and the design of a given system, while preserving its behavior [13].

However, often it is not possible to directly extract to aspects concerns suggested by aspect mining tools [24,2,3]. The reason is that aspect-oriented languages only support the introduction of crosscutting behavior in join points, i.e., well-defined points of the execution of object-oriented systems. For example, in AspectJ join points include methods calls and execution, read and write to fields, exception handlers execution, class initialization, etc. However, in legacy systems we should not expect that crosscutting code is located precisely before, around or after join points, as required by AspectJ. Thus, after the mining of crosscutting concerns and before the beginning of the refactoring task, programmers usually need to transform the base program in order to associate crosscutting statements with parts of the program that can be captured by the pointcuts of an aspect-oriented language [2,3].

Most of the works in the area of aspect-oriented refactoring frequently mention the need of such enabling object-oriented transformations. However, they usually do not provide detailed information about the spectrum of the possible transformations and the frequency that such transformations are required when migrating real-world systems to aspects. For example, in one of the first papers about aspect-oriented refactoring, Monteiro recognizes that “it is sometimes necessary to refactor the base code in order to expose the necessary join points to AspectJ” [23]. However, even in Monteiro more recent papers, this necessity is not investigated in details [24,25]. In a paper from 2001, Murphy et al. affirm that a concern can be easier modularized “if advance work to

* Corresponding author.
E-mail addresses: nassau@pucminas.br (M.N. Malta), mtov@pucminas.br (M.T. de Oliveira Valente).

prepare the software system is undertaken" [26]. However, the recommended preparation only includes encapsulating concerns in entire methods and classes and moving groups of crosscutting statements to the beginning and ends of methods. When proposing a tool and a refactoring methodology for decomposing legacy applications into a set of features, Liu, Batory, and Lengauer mention that rearranging the order of statements may be needed before tangling features can be extracted [22]. However, they have not quantified the frequency that such reorderings are required. Instead, they only mention that "several iterations of this step may be necessary to achieve an acceptable refactoring". Binkley and colleagues have developed the AOP-Migrator tool, an Eclipse plug-in that automates a set of aspect-oriented refactorings [3]. They recognize that "OO transformations represent an important cost in the migration process" towards aspect-oriented systems and present wrap-up statistics on the use of such transformations in the refactoring of four medium-sized Java programs. However, since their emphasis was on the presentation of the refactorings automated by AOP-Migrator, the authors have not devoted too much effort in describing and giving examples about object-oriented transformations.

This paper documents an in-depth investigation about transformations that must be applied to the base code of object-oriented systems, after the identification of crosscutting concerns and before the encapsulation of such concerns in aspects. The contributions of the paper are twofold. First, it describes a collection of object-to-object transformations used to enable the extraction of crosscutting statements to aspects. Similar to catalogs of object-oriented refactorings, the description is illustrated by examples taken from real-world aspect-oriented systems. Second, the paper details results about the use and the importance of such transformations in object-oriented systems that have been migrated to aspects. Such results demonstrate that the transformations investigated in the paper are critical to turn legacy systems ready to aspect-oriented refactorings. For example, the ratio between the number of applied transformations to the number of join points advised by aspects has ranged from 0.15 to 1.33 in the systems considered in our study. We have also concluded that it is very complex to automate the proposed transformations, even if supposing that concerns have been previously located by an aspect mining tool.

The remaining of the paper is organized as follows. We start out describing the identified transformations and giving examples of their use in four Java systems (Section 2). Next, Section 3 quantifies the use of the proposed transformations in such systems and correlates this use with the kind of crosscutting concern being aspectized. In Section 4 we discuss related work and Section 5 presents the conclusions. Finally, Appendix A presents a formal description of the transformations used in the paper.

2. Object-oriented transformations

Following the convention proposed by Binkley and colleagues, an object-oriented transformation is a reorganization of the source code of a given system that preservers its behavior and enables an aspect-oriented refactoring [2,3]. Since OO transformations usually do not improve the internal design of the system, they are not called refactorings. Instead, in this paper the name refactoring is reserved to manipulations that aspectize a given OO code.

We have only considered transformations applied to crosscutting code of Java systems, i.e., code that is tangled and spreaded among the classes of systems implemented in this language. Moreover, it is assumed that the candidated concerns have been previously located in the source code, possibly using an aspect mining tool. Our investigation considers AspectJ as the target aspect language, since it is the most mature and the most widely used aspect language nowadays.[1] For this reason, the proposed OO transformations have the purpose to enable aspect extraction accordingly to the dynamic join point model supported by AspectJ. This model limits the available join points to the following events: method calls or executions, field gets or sets, exception handler executions and class initializations.

The examples used to illustrate the transformations described in this paper have been taken from the following systems:

- Jaccounting[2]: a Web-based business accounting system, which automates invoicing, bills and accounts handling. In this paper, we used as example transformations performed by Binkley and colleagues in order to aspectize JAccounting transaction management code [3].
- JHotdraw[3]: a framework targeting applications for drawing technical and structured graphics. In this paper, we used as example transformations performed by Binkley and colleagues in order to aspectize JHotDraw undo concern [2].
- Prevayler[4]: a persistence system for Java objects. In this paper, we used as example transformations performed by Godil and Jacobsen in order to aspectize the following Prevayler concerns: snapshots, clocks, censoring, replication, persistent logging, and multithreading [14].
- Tomcat[5]: a Web container that supports servlets and JSPs. In this paper, we used as example transformations we have performed in order to aspectize the logging concern from a subset of the Tomcat packages. The subset

[1] Besides AOP/AspectJ, other alternatives can be used in the modularization of crosscutting concerns, including meta-object protocols [20] and mixins [4].

[2] https://jaccounting.dev.java.net.

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

[4] http://www.prevayler.org.

[5] http://tomcat.apache.org.