JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 3, May–June 2009

Guidelines for Enabling the Extraction of Aspects from Existing Object-Oriented Code

Marcelo Nassau Malta, Samuel de Oliveira, Marco Tulio Valente
Institute of Informatics PUC Minas, Brazil

When extracting crosscutting concerns from object-oriented systems to aspects, it is often needed to transform the code in order to enable the application of aspects. Although not extensively documented in the literature, object-oriented transformations are critical to turn legacy systems ready to aspect-oriented refactoring. For this reason, in this paper, we provide a set of guidelines for maintainers and developers interested in preparing object-oriented systems to the aspect technology. Moreover, we present a tool that can help developers to identify the need of object-oriented transformations. We also evaluate the proposed guidelines and supporting tool through two case studies.

1 INTRODUCTION

Aspects represent nowadays the principal technology for the modularization of crosscutting concerns, i.e. concerns that are poorly encapsulated using traditional modularization abstractions, such as procedures and classes. The success of the technology has triggered research in most areas of software engineering, from requirements to tests. Particularly, several works about aspect-oriented refactoring have been conducted, with the purpose to describe how tangled and scattered object-oriented implementations can be moved to equivalent aspects [16, 3, 11, 7].

However, one of the principal obstacles for applying aspect-oriented refactoring techniques is the fact that aspect languages – such as AspectJ [10] – only allow aspects to advise well-defined points in the execution of object-oriented systems, called join points. For example, the dynamic crosscutting model of AspectJ provides support to the following types of join points: method calls and execution, fields read and setting, exception handler execution, and class and object initialization. However, regarding existing systems, it is not conceivable to assume that crosscutting concerns are always implemented in pre-defined locations of the object-oriented code. Thus, after the identification of crosscutting concerns and before starting their extraction to aspects, developers usually need to transform the base code in order to enable the application of aspect-oriented refactorings. These transformations are usually called object-oriented transformations (or just transformations, in the context of this paper) [1, 2, 13].

Despite their importance in the aspectization of object-oriented systems, most works about aspect-oriented refactoring mention the need of object-oriented transformations in a concise way. For this reason, in previous work, we have described

Cite this document as follows: Susanne Cech: *Guidelines for Enabling the Extraction of Aspects from Existing Object-Oriented Code*, in Journal of Object Technology, vol. 8, no. 3, May–June 2009, pages 101–119,
http://www.jot.fm/issues/issues_2009_3/article3

a catalogue of transformations that can be applied to object-oriented systems, after the identification of crosscutting concerns and before the encapsulation of such concerns in aspects [13]. In this paper, we complement our previous research by providing guidelines to assist developers in the application of the proposed catalogue. Our main motivation was the observation that – as proposed – our catalogue was just a "static" description of code transformations commonly used to prepare object-oriented systems in order to extract aspects. However, after an in-depth investigation about the concrete application of these transformations in real-world systems, we have observed that there are general principals and rules behind the application of the documented transformations. Following such principles is fundamental to assure that (i) the correct transformations are always applied, (ii) that transformations are not applied unnecessarily and (iii) that transformations contribute to key attributes of the extracted aspects (such as quantification). For this reason, in this paper we provide detailed information to guide developers when preparing legacy code to aspectization. Moreover, we present an Eclipse plugin that can help developers in the application of such guidelines. We also evaluate the proposed guidelines and the supporting tool through two small-to-medium sized case studies.

The focus of the paper is on dynamic crosscutting concerns, i.e. crosscutting concerns that can be modularized by means of advices [10]. More specifically, we are interested in transformations that have to be applied to existing, object-oriented code in order to enable the extraction of crosscutting concerns to advices. Such object-oriented transformations can be classified as statement reordering and method extraction transformations [13]. Statement reordering transformations prescribe the movement of statements with a crosscutting behavior to before or after statements of the base code whose execution can be intercepted by the pointcut language of AspectJ. On the other hand, method extractions are recommended when the crosscutting code is not before or after a join point, neither can be moved to a join point. In this case, the goal of method extraction is to transform part of the non-crosscutting code that succeeds or precedes the code classified as crosscutting into a method. In this way, the transformation creates a join point shadow whose join points can be intercepted by AspectJ's pointcut language.

The remainder of the paper is organized as follows. Section 2 describes basic guidelines for developers interested in the application of OO transformations. The section presents strategies to decide whether OO transformations are demanded and how to select the most adequated transformations. The next two sections describe more complex transformation strategies, that can improve the join points enabled with the basic transformations. Particularly, Section 3 describes the notion of ad hoc transformations, that can be used to enable the preconditions required by traditional OO transformations. Section 4 presents guidelines to optimize the aspectization of homogeneous concerns. In Section 5, we describe an Eclipse plugin that helps developers to decide in which parts of the base code OO transformations are required. Section 6 describes two case studies about the application of the proposed guidelines and tool. Section 7 presents related work and Section 8 concludes.

2 BASIC GUIDELINES

This section describes guidelines for the application of object-oriented transformations used to enable the extraction of pieces of advice from object-oriented systems. The assumption is that developers have already identified the concerns presenting a crosscutting behavior, possibly with the support of an aspect mining tool [9, 14]. We start presenting an algorithm that can be used to decide whether OO transformation are needed. Next, we present guidelines to select the most adequated transformations.

Deciding whether OO transformations are needed

In order to decide whether OO transformations are required, developers should regard on the following decision algorithm:

1. The initial step is to build a map, called crosscutting map, indicating the locations of the base code containing the implementation of crosscutting concerns. Moreover, we assume that crosscutting corresponds to single method calls. Since transformations are always associated to dynamic crosscutting concerns, this assumption does not impose any limitation to the expressiveness of the proposed guidelines. For example, concerns associated to other statements (assignments, loops etc) or concerns associated to multiple statements must be first extracted to a method, using the Extract Method refactoring [6].

2. For each mapped entry, developers must check whether it appears in the beginning or in the end of the lexical scope of the following elements: a body of a method, constructor, or exception handler. Developers must also evaluate whether the entry appears before or after one of the following elements: a field read, a field write or a method call. If the crosscutting code does not match any of the mentioned conditions, then a transformation is required.

Transforming the base code

Suppose that after executing the previous decision algorithm, developers have figured out that they need to apply an OO transformation. In such cases, they should regard on the following guidelines to decide which transformation to apply.

1. First, developers must try to apply a statement reordering transformation. The most difficult question regarding the application of such transformations is to evaluate their preconditions. In order to help developers in this step, Figure 1 presents the preconditions and the possible code transformations prescribed by statement reordering transformations[1].

[1] A detailed and formal presentation of these transformations is available in [13].

Supposing that cc are crosscutting statements and mtd is a method declarator

(S1) Move code to the beginning of a method body
$\texttt{mtd}\,\{\underbrace{\texttt{st cc}}_{\text{chg}}\,\texttt{st}'\} \Rightarrow \texttt{mtd}\,\{\texttt{cc st st}'\}$

(S2) Move code to the end of a method body
$\texttt{mtd}\,\{\texttt{st}'\ \underbrace{\texttt{cc st}}_{\text{chg}}\} \Rightarrow \texttt{mtd}\,\{\texttt{st}'\ \texttt{st cc }\}$

(S3) Move code to the statement before a return
$\texttt{mtd}\,\{\texttt{st}_1\ \underbrace{\texttt{cc st}_2}_{\text{chg}}\,\texttt{return st}_3\} \Rightarrow \texttt{mtd}\,\{\texttt{st}_1\ \texttt{st}_2\ \texttt{cc return st}_3\}$

(S4) Move code to the statement before a method call
$\texttt{mtd}\,\{\texttt{st}_1\ \underbrace{\texttt{cc st}_2}_{\text{chg}}\ \texttt{m() st}_3\} \Rightarrow \texttt{mtd}\,\{\texttt{st}_1\ \texttt{st}_2\ \texttt{cc m() st}_3\}$

(S5) Move code to the statement after a method call
$\texttt{mtd}\,\{\texttt{st}_1\ \texttt{m()}\ \underbrace{\texttt{st}_2\ \texttt{cc}}_{\text{chg}}\ \texttt{st}_3\} \Rightarrow \texttt{mtd}\,\{\texttt{st}_1\ \texttt{m()}\ \texttt{cc st}_2\ \texttt{st}_3\}$

(S6) Move code to the beginning of a catch block
$\texttt{catch(e)}\,\{\underbrace{\texttt{st cc}}_{\text{chg}}\,\texttt{st}'\} \Rightarrow \texttt{catch(e)}\,\{\texttt{cc st st}'\}$

(S7) Move code to the end of a catch block
$\texttt{catch(e)}\,\{\texttt{st}'\ \underbrace{\texttt{cc st}}_{\text{chg}}\} \Rightarrow \texttt{catch(e)}\,\{\texttt{st}'\ \texttt{st cc }\}$

(S8) Move code to the statement before a field read or write
$\texttt{mtd}\,\{\texttt{st}_1\ \underbrace{\texttt{cc st}_2}_{\text{chg}}\ \texttt{set/get st}_3\} \Rightarrow \texttt{mtd}\,\{\texttt{st}_1\ \texttt{st}_2\ \texttt{cc set/get st}_3\}$

(S9) Move code to the statement after a field read or write
$\texttt{mtd}\,\{\texttt{st}_1\ \texttt{set/get}\ \underbrace{\texttt{st}_2\ \texttt{cc}}_{\text{chg}}\ \texttt{st}_3\} \Rightarrow \texttt{mtd}\,\{\texttt{st}_1\ \texttt{set/get cc st}_2\ \texttt{st}_3\}$

The rules are subject to the following side conditions:

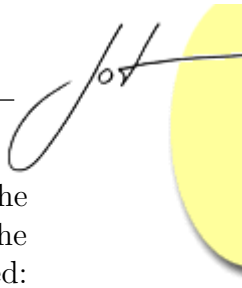
S3: There is no other return statement in mtd
S4-S5: There is no other call to m in mtd
S8-S9: If moving to a set, there is no other set to the same field in mtd. If moving to a get, there is no other get to the same field in mtd

Figure 1: Statement reordering transformations

In this figure, the rules that describe the transformations have the following general form:

$$\ldots\ \underbrace{\texttt{st}_1\ \texttt{st}_2}_{\text{chg}}\ \ldots\ \Rightarrow \ldots\ \texttt{st}_2\ \texttt{st}_1 \ldots$$

The left side of this rule specifies that it must be possible to change the order



of statements $\texttt{st}_1$ and $\texttt{st}_2$. If this precondition is satisfied, the right side of the rule describes the structure of the code after the transformation. Moreover, the following function must be used to decide if two statements can be reordered:

$$\begin{aligned}\mathbf{chg}(\texttt{st}_1, \texttt{st}_2) \equiv\ & \mathbf{def}(\texttt{st}_1) \cap \mathbf{ref}(\texttt{st}_2) = \emptyset\ \wedge\ \mathbf{ref}(\texttt{st}_1) \cap \mathbf{def}(\texttt{st}_2) = \emptyset \wedge \\ & \mathbf{def}(\texttt{st}_1) \cap \mathbf{def}(\texttt{st}_2) = \emptyset \wedge !\mathbf{exit}(\texttt{st}_2)\end{aligned}$$

According to this function, the order of $\texttt{st}_1$ and $\texttt{st}_2$ can be changed when: $\texttt{st}_1$ does not define any variable that is referenced by $\texttt{st}_2$; $\texttt{st}_1$ does not reference any variable defined by $\texttt{st}_2$; both $\texttt{st}_1$ and $\texttt{st}_2$ do not define any variable in common; and $\texttt{st}_2$ does not affect the execution of the program in a way that precludes the execution of $\texttt{st}_1$ (for example, throwing an exception or executing a `return`).

2. In case statement reordering cannot be applied, developers must evaluate the application of method extraction transformations. Figure 2 presents the preconditions and the possible code transformations prescribed by method extractions [13]. In this figure, the rules have the following form:

$$\ldots\ \underbrace{\texttt{st}_1\ \texttt{st}_2\ \ldots \texttt{st}_n}_{\texttt{m}}\ \ldots\ \Rightarrow \ldots\ \texttt{m(p)}\ \ldots$$

First, this rule requires the extraction of statements $\texttt{st}_1\ \texttt{st}_2\ \ldots \texttt{st}_n$ to a method named `m`, which is then called in the right side.

3 AD HOC TRANSFORMATIONS

Before developers conclude they cannot apply a statement reordering transformation because its preconditions are not satisfied, they can attempt an ad hoc transformation. More specifically, the purpose of ad hoc transformations is to restructure the base code in order to enable the application of another transformation [2]. Since ad hoc transformations prescribe changes that are very specific to a given concern and code base, it is very difficult to formalize their mechanisms. However, they usually rely on temporary variables or in extra statement reordering to preserve the state of the system or to reestablish its normal execution flow. In order to illustrate the application of an ad hoc transformation, suppose the following fragment of code:

```
Display.save(dirty);              // aspect (requires dirty==true)
dirty= false;
draw();                           // join point
```

Supposing cc are crosscutting statements; mtd is a method declarator; m or m' is the name of the extracted method; v is a local variable.

(M1) Extract method with code that precedes a crosscutting concern

$$\mathtt{st}_1 \underbrace{\mathtt{st}}_{\mathtt{m}} \mathtt{cc}\ \mathtt{st}_2 \Rightarrow \mathtt{st}_1\ \mathtt{m(p)}\ \mathtt{cc}\ \mathtt{st}_2$$

(M2) Extract method with code that succeeds a crosscutting concern

$$\mathtt{st}_1\ \mathtt{cc}\ \underbrace{\mathtt{st}}_{\mathtt{m}}\ \mathtt{st}_2 \Rightarrow \mathtt{st}_1\ \mathtt{cc}\ \mathtt{m(p)}\ \mathtt{st}_2$$

(M3) Extract method with code that precedes and succeeds a crosscutting concern

$$\mathtt{st}_1 \underbrace{\mathtt{st}}_{\mathtt{m}}\ \mathtt{cc}\ \underbrace{\mathtt{st}'}_{\mathtt{m}}\ \mathtt{st}_2 \Rightarrow \mathtt{st}_1\ \mathtt{cc}\ \mathtt{m(p)}\ \mathtt{st}_2$$

(M4) Extract method including a try-catch-finally block

$$\mathtt{st}_1\ \underbrace{\mathtt{try\{st'\}\ catch\{cc_1\}\ finally\{cc_2\}}}_{\mathtt{m}}\ \mathtt{st}_2 \Rightarrow \mathtt{st}_1\ \mathtt{m(p)}\ \mathtt{st}_2$$

(M5) Extract method when withincode can not determine a join point

$$\mathtt{mtd}\ \{\mathtt{st}_1\ \underbrace{\mathtt{m(p)}}_{\mathtt{m'}}\ \mathtt{cc}\ \mathtt{st}_2\} \Rightarrow \mathtt{mtd}\ \{\mathtt{st}_1\ \mathtt{m'(p')}\ \mathtt{cc}\ \mathtt{st}_2\}$$

The rules are subject to the following side conditions:

M5: There is another call to m in $\mathtt{st}_1$ or in $\mathtt{st}_2$.

Figure 2: Method extraction transformations

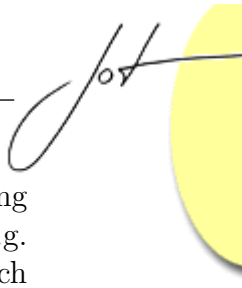
Suppose we want to apply transformation S4 to move the save call to just before the draw call. However, supposing that save reads the dirty variable, the application of S4 would not be possible (as prescribed by its precondition). Thus, an ad hoc solution to bypass this restriction is to make the dirty assignment succeed the call to the draw method:

```
Display.save(dirty);              // aspect
draw();                           // join point
dirty= false;
```

In summary, ad hoc transformations depend on developers ability to preserve the original semantics of the program, while enabling the preconditions required by the target transformation.

4 HOMOGENEOUS CONCERNS

When implemented using object-oriented languages, homogeneous crosscutting concerns require the presence of the same block of code in several parts of the system [4]. Probably, homogeneous concerns are the most adequated type of concern for modularization using aspects. The reason is that their code can be encapsulated in a



single advice. On the other hand, in order to simplify their implementation using aspects, homogeneous concerns should appear consistently in the source code (e.g. always before/after calling the same method, always at the beginning/end of catch blocks etc). When this level of consistency is achieved, it is more simple to specify a single pointcut expression and a single type of advice (i.e. before, after or around) to implement the concern.

In this section we first propose an equivalence relation to classify a concern as homogeneous. Next, using the defined relation, we extend the basic guidelines, described in Section 2, to handle the aspectization of homogeneous concerns.

Definition: We will assume that the scattered calls to be classified as homogeneous or heterogeneous correspond to single-argument method calls. However, a straightforward extension to the following definition can be derived to methods with multiple arguments.

Suppose the following calls to a given method m: $t_1.m(arg_1)$ and $t_2.m(arg_2)$, where t_i denotes the target and arg_i denotes the argument of the calls ($i = 1$ or $i = 2$). These calls are homogeneous when both of the following conditions holds:

- t_1 and t_2 denote the same class (or classes having a common superclass) or fields having the same type (or that are derived from a common type).
- arg_1 and arg_2 are the same constant value or fields having the same type (or that are derived from a common type).

Example: Suppose the calls to `start` and `log` in the following classes:

```
class A {
  Transaction tx;
  void foo(){
    tx.start(1);
    ....
    Logger.log("finished");
    ....
  }
}
```

```
class B {
  Transaction tx;
  void bar(){
    tx.start(1);
    ....
    Logger.log("panic");
    ....
  }
}
```

The `start` calls are homogeneous because: (i) their target represent fields that have the same type (`Transaction`); (ii) their arguments are the same integer constant. On the other hand, the `log` calls are not homogeneous, because although they rely on a static method of the same class (`Logger`), their arguments are distinct strings.

Equivalence Relation: In the following equations, we rely on the notation $\mathtt{t_i.m(arg_i)} \equiv \mathtt{t_j.m(arg_j)}$ to denote calls that are homogeneous according to the previous definition. More specifically, $\equiv$ represents an equivalence relation (i.e. a relation that is reflexive, symmetric, and transitive). The equivalence classes of a method call in the form `t.m(arg)`, denoted by $[\mathtt{t.m(arg)}]_{\equiv}$, is defined as:

$$[\mathtt{t.m(arg)}]_{\equiv} = \{\, \mathtt{t'.m(arg')} \in M \mid \mathtt{t'.m(arg')} \equiv \mathtt{t.m(arg)} \,\}$$

In this definition, M is the set of all scattered calls to `m` in an object-oriented system. Based on this definition, M is a considered a fully homogeneous concern when:

$$\forall\ \mathtt{t.m(arg)} \in M, [\mathtt{t.m(arg)}]_{\equiv} = M$$

In other words, a concern is fully homogeneous when any possible pair of calls implementing this concern are homogeneous, according to the relation $\equiv$. On the other hand, M is a considered a fully heterogeneous concern when:

$$\forall\ \mathtt{t.m(arg)} \in M, [\mathtt{t.m(arg)}]_{\equiv} = \{\ \mathtt{t.m(arg)}\ \}$$

In other words, a concern is fully heterogeneous when the equivalence class of a given scattered call contains only itself.

Guidelines: In order to optimize the aspectization of homogeneous concerns, the following rules complement the basic guidelines described in Section 2:

1. Suppose the crosscutting map described in Section 2. After building this map, developers must partition the mapped calls in equivalence classes, according to the $\equiv$ relation.

2. The same OO transformation must be applied to the members of a given equivalence class. To achieve this goal, developers must also consider the use of ad hoc transformations. Assuming this guideline is followed, the number of equivalence classes generated in the previous step indicates the number of advices needed to encapsulate the implementation of the mapped concern.

The rationale for such guidelines is straightforward. As defined, homogeneous crosscutting concerns can be modularized by a single advice; but this is facilitated if the concern is implemented in a consistent way in the code base. Moreover, the guidelines establish a one-to-one relation between the proposed equivalence classes and the advices used to implement homogeneous concerns.

5 TRANSFORMATIONMAPPER

In this section we describe a prototype tool, called TransformationMapper, that provides partial support to the guidelines proposed in the previous sections. The TransformationMapper is an extension of the ConcernMapper Eclipse-based plugin, proposed by Robillard et al. to logically reorganize the code of a software system in terms of high-level abstractions called concerns [19]. More specifically, the system provides alternative views about the modularity of a software system, without requiring developers to change its source code (as happen with aspects). This characteristic of the plugin matches our initial guideline to build a logical map that can guide the application of OO transformations. Moreover, the ConcernMapper was designed as a platform for experimenting with advanced separation of concerns mechanisms, which makes more simple the derivation of extensions from its core implementation.

In order to start using the plugin developers should first create a crosscutting map, which in our extension corresponds to the concept of concern in the original ConcernMapper implementation. Next, developers should drag-and-drop to the crosscutting map the methods that have been previously classified as having a crosscutting behavior. The TransformationMapper then performs two key operations:

- First, the tool automatically locates calls to the selected method in the source code, using the search engine provided by the Eclipse platform. It also automatically inserts each located call in the crosscutting map. Therefore, this feature of the plugin supports the first step of the guidelines for deciding whether OO transformations are needed described in Section 2.
- Second, for each call inserted in the crosscutting map, the tool informs whether its aspectization demands an OO transformation[2]. Therefore, this feature of the plugin supports the second step of the guidelines presented in Section 2.

Figure 3 illustrates the mentioned operations. This figure shows a crosscutting map describing the logging concern of a given system. More specifically, this concern is implemented by two different methods: `debug(Object, Throwable)` and `debug(Object)`. Calls to the second method are presented in many classes of the base system, such as in `EventDispatcherImpl`. Particularly, there are five calls in the constructor of this class (in lines 53, 63, 64, 65 and 78). The calls performed in lines 53 and 78 can be directly extracted to advices, i.e. these calls are located in static locations of the base program that can be instrumented by AspectJ's pointcut language. On the other hand, the calls performed in lines 63, 64 and 65 demand the application of OO transformations in order to enable their aspectization.

[2]The only exception in this case is Transformation M4. Since this transformation prescribes the extraction of a method containing the implementation of crosscutting concerns related to exception handling, its application is handled by our initial assumption that crosscutting code always correspond to method calls. Thus, in this particular case, the TransformationMapper just detects that the required transformation was applied when building the crosscutting map.

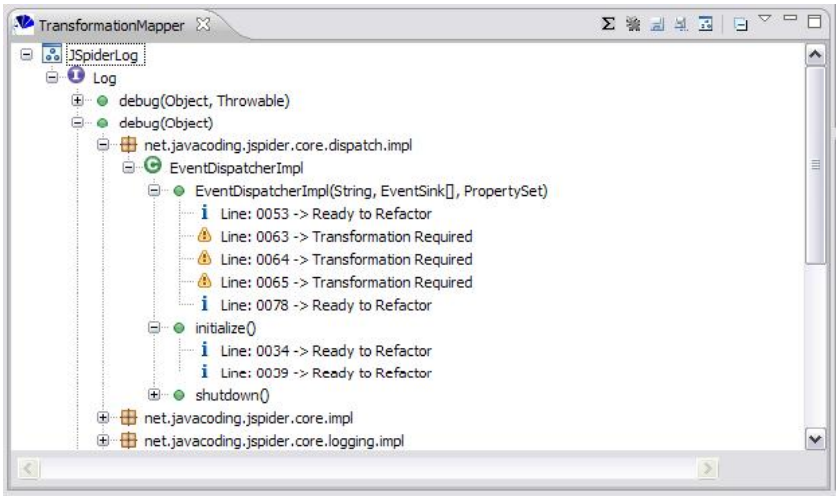
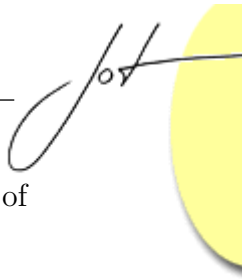


Figure 3: TransformationMapper

The TransformationMapper also automates the construction of the equivalence classes for the entries of a given crosscutting map, according to the $\equiv$ relation defined in Section 4. As described, this is fundamental to optimize the aspectization of homogeneous concerns. The tool also provides general statistics about the crosscutting map (such as total number of entries, total number of transformations required, total number of homogeneity classes generated etc).

It is worth to mention that the TransformationMapper is a supporting tool, in the sense that it does not have the purpose to completely automate the application of OO transformations. In fact, fully automating the proposed transformations is a challenging task [2, 13]. Particularly, the system does not select and apply the transformations (according to the guidelines of Section 2). This would require for example the verification of the preconditions of statement reordering transformations. However, this verification demands the availability of system-wide dependency information, including dependencies originated from external sources such as SGBDs, remote objects etc. Usually, such dependencies are not considered by tools in charge of building dependency graphs or slicers for Java programs [8, 5]. In addition to that, the TransformationMapper does not provide support to ad hoc transformations, since it is very complex to provide even a step-by-step description of the possible operations required by such transformations.

Implementation Details: The implementation of the system reuses components provided by the ConcernMapper to build and organize logical views of a software system. The system also relies on two key components of the Eclipse platform. The Search Engine is used to locate the presence of crosscutting calls in the codebase. Such calls are then automatically inserted in the tree that represents the crosscutting map. The Abstract Syntax Tree (AST) exported by the Eclipse platform is used to decide whether OO transformations are required. By transversing the AST, it is possible to verify if a mapped crosscutting call is located in a static location of



the base program that can be instrumented by the dynamic crosscutting model of AspectJ.

6 EVALUATION

In this section, we describe our experience in applying the proposed guidelines in the preparation of the following systems to aspectization:

- JSpider[3], a 9019 LOC Web robot engine that supports downloading and validation of web pages. In JSpider, we decided to apply the proposed guidelines to enable the aspectization of the system's logging concern. In the original, object-oriented implementation of the system, this is clearly a crosscutting concern, requiring developers to call methods from the logging API in several parts of the system.
- JAccounting[4], a 6526 LOC business accounting system, which automates invoicing, bills and accounts handling. JAccounting relies on the Hibernate framework for persistence and transaction control. Particularly, transactions are implemented by calling transaction services, such as `commit` and `rollback`, in many locations of the OO code.

The investigated systems are small-to-medium OO systems, publicly available for downloading from open source code repositories. More important, they have been previously used by Binkley et al. to validate an aspect-oriented refactoring tool [1, 2]. This fact provide us a baseline for comparison, i.e. we can compare transformations that follow the guidelines proposed in this paper – and that have been applied with the support of the TransformationMapper tool – with transformations independently performed by Binkley and colleagues without tool support.

JSpider

First, using the TransformationMapper, we have constructed a crosscutting map pointing to the locations of the base code implementing logging concerns. For this purpose, we informed that the `log(Object)` is the method responsible for logging in the system. The TransformationMapper has then located 245 calls of this method in the codebase. Next, these calls have been inserted to the crosscutting map. From these 245 entries, the system has then indicated that 22 entries require the application of OO transformations (i.e. 9% of the entries).

We have also relied on the TransformationMapper to construct the equivalence classes used to distinguish between homogenous and heterogeneous concerns. The

[3] http://j-spider.sourceforge.net.
[4] https://jaccounting.dev.java.net.

system has identified that the mapped calls can be divided in 229 equivalence classes, according to the $\equiv$ relation. From these 229 classes, 219 have just one method call, six classes have just two method calls, three classes have three method calls and one class has five method calls. Therefore, the mapped calls have an inherently heterogeneous behavior (i.e. more than 95% of the calls are equivalent to just themselves). This is an interesting result because earlier studies about the benefits of AOP often mention logging as a homogeneous concern [4]. However, in JSpider it is common to have very specific log messages in each point of the code, in order to reflect the exact behavior of this part of the system. According to the definition of homogeneity from Section 4, these particular messages turn the log calls distinct from each other and thus the concern should be considered heterogeneous.

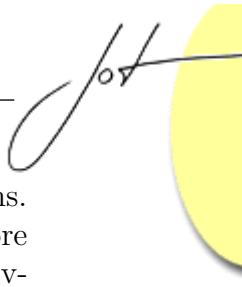
Table 1 compares and correlates the transformations applied in the aspectization performed by Binkley et al. with the transformations indicated by the TransformationMapper. Binkley et al. have performed a total of 36 transformations in the code; on the other hand, as we have mentioned, the TransformationMapper has recommended a total of 22 transformations. However, as we can see in the third column of this table, many transformations performed by Binkley and colleagues were not really necessary (and for this reason have not been indicated by the TransformationMapper). Indeed, in order to enable the aspectization of the logging code in JSpider, 24 transformations are effectively needed. Thus, Binkley et al. have performed 12 transformations without necessity. On the other hand, from the 24 mandatory transformations, the TransformationMapper has not detected the need of only two ad hoc transformations.

	Binkley	**TM**	**Mandatory**
Transformation S1	3	2	2
Transformation M1	6	2	2
Transformation M2	16	10	10
Transformation M3	3	2	2
Transformation M4	5	5	5
Transformation M5	1	1	1
Ad hoc	2	0	2
Total	36	22	24

Table 1: Transformations performed by Binkley et al., transformations suggested by the TransformationMapper (TM), and transformations effectively required, regarding the aspectization of JSpider's logging concern.

JAccounting

In order to evaluate the need of OO transformations in the aspectization of transactions in the JAccounting system, we have included three method calls in the crosscutting map: `beginTransaction`, `commit`, and `rollback`. Next, we requested



the calculation of the equivalence classes used to characterize homogenous concerns. The result was that such methods represent fully homogeneous concerns. More specifically, all the calls to `beginTransaction` have been included in a single equivalence class. The same happens to the `commit` and `rollback` calls.

When handling homogeneous concerns, the proposed guidelines recommend that homogenous calls should happen at equivalent locations of the base code. This guideline has been followed in the refactorization performed by Binkley et al. Particularly, they have decided to move the `beginTransaction` calls to just after the join points where database sessions are opened, as illustrated by the following example:

```
Session sess= getSessionFactory().openSession();
Transaction tx= sess.beginTransaction();         // aspect
```

For this purpose, they have applied 14 transformations S5 (move code to the statement after a method call; in this case, `openSession`).

Moreover, they have decided to move the `commit` calls to just before the join points where database sessions are closed, as illustrated by in the following example:

```
 1: catch (Exception e) {
 2:   tx.rollback();  // aspect
 3:   tx= null;       // ad hoc transformation
 4:   throw e;
 5: }
 6: finally {
 7:   if (tx != null)   // ad hoc transformation
 8:      tx.commit();   // aspect
 9:   sess.close();
10: }
```

For this purpose, they have applied 15 transformations S4 (move code to the statement before a method call; in this case, `sess.close()` in line 9). Moreover, in order to enable the application of this transformation, 30 ad hoc transformations have been applied: 15 transformations to check if `tx` is `null` before calling `commit` (line 7), and 15 transformations to assign `null` to `tx` in case a `rollback` has been called (line 3). In this way, they guarantee that if rollback is called, commit will not be called, and vice-versa.

Discussion

The following observations can be derived from the case studies:

- As mentioned in our previous studies, transformations are important operations in the aspectization of object-oriented systems [13]. For example, 10% of the logging code in JSpider has demanded the application of enabling OO

transformations. In JAccounting, this number was even superior. The legacy, object-oriented version of this system has 45 scattered and tangled calls related to transaction handling. In order to enable the aspectization of these calls, 60 transformations have been applied to the legacy code (including ad hoc transformations).

- The proposed guidelines – and to a great degree the TransformationMapper plugin – have effectively contributed to detect the need of statement reordering and method extraction transformations. Also, the guidelines have contributed to avoid unnecessary transformations, as was observed in the preparation of the JSpider by Binkley and others.

- As demonstrated in the JAccouting case study, the proposed guidelines for handling homogeneous concerns can effectively contribute to the extraction of pieces of advices presenting key benefits normally associated to aspect-oriented programming, such as quantification. However, in such cases the ability of the programmer to apply ad hoc transformations may also be fundamental to achieve the desired properties.

- It was very simple to construct the crosscutting map of the evaluated systems. More important, the crosscutting map provides important information that can be used by developers to evaluate the overall effort involved in the aspectization of a given software system. For example, the tool provides information about the number of scattered and tangled calls, the number of transformations that must be applied to the codebase, and the number of advices that can be extracted. More important, such information is provided before the implementation of any AspectJ code. Thus, it represents a valuable asset to developers interested in evaluating the possible benefits of applying aspect-oriented techniques to existing systems.

7 RELATED WORK

Our decision to provide guidance regarding enabling, object-oriented transformations was motivated by the lack of details about such transformations in works about aspect-oriented refactoring.

For example, in one of the first papers about AO refactoring, Monteiro recognizes that "it is sometimes necessary to refactor the base code in order to expose the necessary join points to AspectJ" [15]. However, the author has not documented such refactorings in details, even in further work [16]. More recently, Monteiro and Fernandes have proposed a refactoring process to guide the transformation of a Java source code base into a functionally equivalent AspectJ source code base [17]. The proposed refactoring process has three phases: in the first phase crosscutting features are extracted to aspects; the second phase aims to improve the internal structure of the extracted aspects, by removing internal duplication; and the third phase targets

the generalization of common code in super-aspects. However, the description of the proposed refactoring process does not mention the need of enabling, object-oriented transformations. For example, in order to extract part of a method to an advice, the authors simply suggest to "create a pointcut capturing the appropriate joinpoint and context and move the code fragment to an advice based on the pointcut", without considering the limitations of AspectJ to capture particular join points.

Binkley and colleagues have developed the AOP-Migrator tool, an Eclipse plugin that automates six refactorings commonly used to support migration from OOP to AOP [1, 2]. The AOP-Migrator's authors recognize that "OO transformations represent an important cost in the migration process" towards aspect-oriented systems. However, since their emphasis was on the presentation of the refactorings automated by AOP-Migrator, the authors have not devoted much effort in analyzing object-oriented transformations.

Murphy et al. suggest that a concern can be easier modularized "if advance work to prepare the software system is undertaken" [18]. 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 [12]. However, they have not documented such rearrangements. Instead, they only mention that "several iterations of this step may be necessary to achieve an acceptable refactoring".

Yuen and Robilliard suggest the existence of an important gap between aspect mining and aspect refactoring tools due to subtle variations in the implementation of crosscutting concerns in legacy systems [21]. For example, they observed that transaction management does not present a consistent behavior in the system used as case study in their research, which suggest the need of statement reordering and possibly ad hoc transformations (as was the case of transaction handling in JAccounting described in Section 6).

Marin et al. have proposed the FINT aspect mining tool, that relies in a fan-in analysis to discover aspects in legacy systems [14]. Basically, their approach looks for methods that are called from many different locations (i.e. methods that have a high fan-in value). Therefore, this technique is particularly useful to discover potential method calls to be inserted in the crosscutting map supported by the TransformationMapper tool. Wloka et al. have presented a refactoring approach and its supporting tool, called SoothSayert, that implements automated adjustments in pointcuts affected by changes in the base code [20]. In this way, they investigated a problem that happens after the extraction of aspects, supposing that such systems are in constant evolution. In the future, we can envision an aspect-oriented refactoring environment that integrates an aspect mining tool (such as FINT), a transformation support tool (such as TransformationMapper), a tool to extract aspects from legacy code (such as AOP-Migrator) and a tool to preserve semantics

properties of pointcuts in face of evolving base code (such as SoothSayer).

8 CONCLUSIONS

In order to provide an aspect-oriented implementation for crosscutting concerns presented in existing, object-oriented systems, developers should: (i) identify crosscutting concerns in the base code; (ii) decide which identified crosscutting concern is worth to refactor using aspect-oriented techniques and languages; (iii) transform the object-oriented code, in order to enable the aspectization of concerns located in non-advisable parts of the base code; (iv) extract the crosscutting code to aspects. Regarding the four mentioned steps, the application of OO transformations has certainly been the one less studied in the literature about aspect-oriented refactoring. For this reason, we have provided in this paper detailed information to assist developers when preparing existing, object-oriented code to aspectization. Moreover, we have presented the TransformationMapper tool, an Eclipse plugin that provides partial support to the proposed guidelines. This system is still a research prototype. However, its current version is available upon request from the authors.

The TransformationMapper – and the guidelines recommended in the paper - have been validated through two small-to-medium case studies, involving the aspectization of both heterogeneous concerns (as was the case of logging in the JSpider system) and homogeneous concerns (as was the case of transactions in the JAccounting case study). Based on the experience gained with the case studies, we have concluded that the proposed guidelines can contribute to detect mandatory, object-oriented transformations and to avoid unnecessary ones. Moreover, they also contribute to key properties of the extracted aspects, such as quantification.

In the near future, we intend to apply the proposed guidelines and the TransformationMapper to new case studies. We also have plans to improve the TransformationMapper with more features and metrics that can provide quantitative information about the benefits of using aspect-oriented languages in legacy systems.

Acknowledgments: This research was supported by a grant from FAPEMIG, process PPM-CEX-APQ 4543-5. We would like to thank David Binkley and Mariano Ceccato for providing us the source code of the aspect-oriented version of the JSpider and JAccounting systems.

REFERENCES

[1] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 27–36, 2005.

[2] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions Software Engineering*, 32(9):698–717, 2006.

[3] Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 123–134, 2005.

[4] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In *3rd International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[5] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *22th International Conference on Software Engineering (ICSE)*, pages 439–448, 2000.

[6] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 135–146, 2005.

[8] Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. Kaveri: Delivering the Indus Java program slicer to Eclipse. In *Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *Lecture Notes in Computer Science*, pages 269–272, 2005.

[9] Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4:145–164, 2007.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*, pages 327–355. Springer Verlag, 2001.

[11] Ramnivas Laddad. Aspect-oriented refactoring. TheServerSide.com, 2003.

[12] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[13] Marcelo Nassau Malta and Marco Tulio de Oliveira Valente. Object-oriented transformations for extracting aspects. *Information and Software Technology*, pages 1–12, 2008 (in press).

[14] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1), 2007.

[15] Miguel P. Monteiro and João M. Fernandes. Some thoughts on refactoring objects to aspects. In *VIII Jornadas de Ingeniera de Software y Bases de Datos (JISBD)*, 2003.

[16] Miguel P. Monteiro and João M. Fernandes. Towards a catalogue of refactorings and code smells for AspectJ. *Transactions on Aspect-Oriented Software Development*, 3880:214–258, 2006.

[17] Miguel P. Monteiro and João M. Fernandes. An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms. *Software Practice and Experience*, 38(4):361–396, 2008.

[18] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *23rd International Conference on Software Engineering (ICSE)*, pages 275–284, 2001.

[19] Martin P. Robillard and Frederic Weigand-Warr. ConcernMapper: Simple view-based separation of scattered concerns. In *OOPSLA Eclipse Technology Exchange Workshop (ETX)*, pages 65–69, 2005.

[20] Jan Wloka, Robert Hirschfeld, and Joachim Hänsel. Tool-supported refactoring of aspect-oriented programs. In *7th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 132–143, 2008.

[21] Isaac Yuen and Martin P. Robillard. Bridging the gap between aspect mining and refactoring. In *AOSD Workshop on Linking Aspect Technology and Evolution*, 2007.

ABOUT THE AUTHORS

Marcelo Nassau Malta is a software architect at PUC Minas (Brazil), where he received his MSc degree in Computer Science. Contact him at nassau@pucminas.br.

Samuel Domingues is a Computer Science undergraduate student at PUC Minas (Brazil). He can be reached at samueldro@gmail.com.

Marco Tulio Valente is an associate professor at the Institute of Informatics at PUC Minas (Brazil). He received his PhD in Computer Science from Federal University of Minas Gerais (Brazil). Contact him at mtov@pucminas.br.