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.

selected includes classes from the `org.apache.tomcat.*` packages (around 45 KLOC from a total of 152 KLOC of the complete system).

The transformations investigated in the paper are divided in two groups: statement reordering and method extraction. The following subsections describe such groups of transformations, giving examples of their use in the mentioned systems. Appendix A provides a formal definition of the proposed transformations.

2.1. Statement reordering

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 captured by the pointcut descriptor language of AspectJ. A list of the seventeen transformations included in this group is showed in Table 1.

The first two transformations in Table 1 prescribe moving statements classified as crosscutting to the beginning or to the end of a method body. The purpose is to enable the modularization of such statements using a pointcut of type `execution` associated to a `before` or `after returning` advice. The third transformation prescribes moving a crosscutting statement to just before a `return`, in order to enable its extraction using an `after returning` advice. Transformations 4–7 determine the movement of crosscutting code to the statement before or after the call of a method or constructor. The purpose is to enable the aspectization of such statements using a pointcut of type `call`, associated to a `before` or `after returning` advice. Transformations 8–9 prescribe moving code to the beginning or to the end of an exception handler block. The intention is to support refactoring of this code using a pointcut of type `handler`. Transformations 10–13 determine moving code to before or after a field read or write. The purpose is to make possible the modularization of this code using a pointcut of type `set` or `get`. Transformations 14 and 15 specify the movement of crosscutting code to the initialization block of a class, in order to permit its extraction using a `staticinitialization` pointcut.

Transformation 16 proposes merging separated crosscutting statements into a single block of statements that can be modularized in an advice. Transformation 17 proposes breaking an expression with tangled crosscutting concerns in two expressions: one in charge of evaluating just the crosscutting concern and the other in charge of evaluating the non-crosscutting behavior. In this way, the former expression can be refactored more easily.

Table 1
Statement reordering transformations

	Description
T1	Move code to the beginning of a method
T2	Move code to the end of a method
T3	Move code to the statement before a return
T4	Move code to the statement before a method call
T5	Move code to the statement after a method call
T6	Move code to the statement before a constructor call
T7	Move code to the statement after a constructor call
T8	Move code to the beginning of a catch block
T9	Move code to the end of a catch block
T10	Move code to the statement before a field read
T11	Move code to the statement after a field read
T12	Move code to the statement before a field write
T13	Move code to the statement after a field write
T14	Move code to the beginning of a class initialization block
T15	Move code to the end of a class initialization block
T16	Merge crosscutting statements into a single block
T17	Untangle crosscutting code from an expression

2.1.1. Examples

The example of Fig. 1 describes the application of the Transformation 1 in the aspectization of the JAccounting system. In this example, the crosscutting statement in charging of starting a transaction has been moved to the beginning of the method block. In this way, a *Extract Beginning of Method* refactoring can be applied [3]. Fig. 2 describes an example of using Transformation 17 in JHotDraw. In this example, the expression originally returned by the method was broken in two subexpressions, whose values were stored in local variables. In this way, the code associated to the JHotDraw undo functionality was restricted to a single expression, enabling a refactoring of type *Extract Pre-Return* [3].

2.1.2. Ad hoc transformations

Statement reorderings should be the first transformations applied when extracting crosscutting statements to aspects. Nevertheless, depending on the program and the concern to be refactored, it is not always possible to change the order of some statements. For example, suppose the following application of Transformation 1:

$\texttt{void foo() \{A;B;...\}} \Rightarrow \texttt{void foo() \{B;A;...\}}$

In order to enable the application of this transformation, the following four preconditions should be met: (i) `B` must not change any state value that is accessed by `A` (including local variables, fields, persistent data, etc.); (ii) `A` must not change any state value that is read by `B`; (iii) both `A` and `B` must not change the value of the same state value; (iv) `B` should not change the execution of the program in a way that precludes the execution of `A` (for example, throwing an exception or executing a `return`). A detailed description of the preconditions required by the proposed transformations is included in Appendix A.

When the preconditions required by a transformation are not accomplished, the programmer can attempt so called ad hoc transformations [3]. The purpose of these transformations is to restore the expected state and the execution flow of the system after the application of a statement reordering transformation. Usually, ad hoc transformations prescribe changes that are very specific to each concern and code base. For example, they can rely

```
Account createInternalAccount(Session sess, ...) throws ... {
 Account account= null;
 net.sf.hibernate.Transaction tx= null;    // aspect
 try {
   tx = sess.beginTransaction();           // aspect
```

⇓

```
Account createInternalAccount(Session sess, ...) throws ... {
 net.sf.hibernate.Transaction tx= null;    // aspect
 tx= sess.beginTransaction();              // aspect
 Account account= null;
 try {
```

Fig. 1. Code before and after the application of Transformation 1 in method of the JAccounting system. Comments indicate the code classified as having a crosscutting behavior.

```
Tool createDragTracker(Figure f) {
   return new UndoableTool(                        // aspect
             new DragTracker(editor(), f));
}
```

⇓

```
Tool createDragTracker (Figure f) {
   DragTracker tmp1= new DragTracker(editor(), f);
   UndoableTool tmp2= new UndoableTool(tmp1);    // aspect
   return tmp2;
}
```

Fig. 2. Example of the application of Transformation 17 in the JHotDraw system.

on temporary variables to save the state of the system or to reestablish its normal execution flow.

Since ad hoc transformation are very dependent from a given base system, this paper does not include a catalog for such transformations. However, Fig. 3 shows an example of the application of an ad hoc transformation in the modularization of the transaction handling concern in the JAccounting system. In this example, Transformation 4 was initially applied to move the `commit` statement to just before the statement that closes the database session. However, this transformation has an impact in the normal execution flow of the system: in the original code, the `commit` is called only when the `try` block executes without any statement throwing an exception; in the transformed code, since the `commit` has been moved to the `finally` block, it is also executed even when an exception is raised. In order to reestablish the original control flow of the system, a temporary variable `tx` is used to indicate if the `catch` block was executed (`tx==null`) or not. Thus, the `commit` is now only called when `tx` is not `null`.

```
 try { .....
    tx.commit();                 // aspect
 }
 catch (Exception e) { ... }
 finally {
    sess.close();
 }
```

⇓

```
 try { ..... }
 catch (Exception e) {  ...  }
 finally {
    tx.commit();                 // aspect
    sess.close();
 }
```

⇓

```
 try { ..... }
 catch (Exception e) { ...; tx= null; ... }
 finally {
    if (tx != null) tx.commit();   // aspect
    sess.close();
 }
```

Fig. 3. Example of the application of Transformation 4 in the JAccounting system, followed by the application of an ad hoc transformation.

2.2. Method extraction

Method extraction is one of the most common object-oriented refactorings [13]. It is used to turn a fragment of code into a method, facilitating its understanding and reuse. In the context of aspect-oriented refactorings, such transformations are recommended when the crosscutting code is not before or after a join point, neither can be moved to a join point, as prescribed by Transformations 1–17. 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 that can be captured using a pointcut of type `call`.

Table 2 describes the transformations included in this group. Transformations 18–20 recommend extracting a method including the code that precedes and/or succeeds statements classified as crosscutting. Transformation 21 prescribes the extraction of a method containing the statements of a `try-catch-finally` block. The purpose is to support the modularization of exception handling code by means of an `around` advice. Similarly, Transformation 22 defines the extraction of a method containing the code of a `synchronized` statement. Transformation 23 recommends the extraction of a method when the designator `withincode` cannot disambiguate the join points captured by a given pointcut, particularly when multiple, syntactically equivalent join points exist in the lexical scope a method. Transformation 24 is useful when an aspect needs to obtain the value of local variables of the base code. Since AspectJ does not consider as join points assignments to local variables, the solution consists in transforming such operations into method calls.

Table 2
Method extraction transformations

	Description
T18	Extract method with code that precedes a crosscutting concern
T19	Extract method with code that succeeds a crosscutting concern
T20	Extract method with code that precedes and succeeds a crosscutting concern
T21	Extract method including a `try-catch-finally` block
T22	Extract method including a `synchronized` statement
T23	Extract method when `withincode` cannot determine a join point
T24	Extract method containing code that sets the value of a local variable

2.2.1. Examples

Fig. 4 shows an example of the application of Transformation 18 in the Tomcat system. After the transformation, a pointcut can be used to capture the call of the extracted method. The logging command can then be refactored to an `after` advice associated to this pointcut. Fig. 5 exemplifies the application of Transformation 21 to extract a method containing a `try-catch-finally` block tangled in a method of the Prevayler system. After the application of this transformation, the *Extract Exception Handling* refactoring can be used to modularize the exception handling code [3]. Fig. 6 describes an example of the use of Transformation 23 in the Prevayler system. Since there are two calls to `readObject` in the method described in this figure, the call preceding the clock crosscutting concern was extracted to the `getTimeStamp` method. In this way, this concern can now be refactored to an `after` advice associated to a pointcut that intercepts only calls to the extracted method.

2.2.2. Limitations

Method extractions have the potential to enable join points in many parts of a system. However, the extracted code can present characteristics that difficult the application of such transformations, such as access to local variables or multiple exit points. In the case of accessing local variables, a recommended solution is to apply the object-oriented refactoring known as *Replace Method with Method Object* [13]. In the case of multiple exits (for example, the existence of a `return` statement in the extracted method), the solution can rely on a flag to indicate a normal exit or an exit due to a `return` statement.

3. Case study

This section presents the results of a study about the use of the described transformations in the aspectization of JAccounting, JHotDraw, Prevayler, and Tomcat systems.

```
void processCookieHeader(byte bytes[], int off, int len ) { ...
  if (bytes[startValue]=='1' && endValue==startValue+1 ) {
     version= 1;
     if(log.isDebugEnabled()) log("Found version=1"); // aspect
  }  ...
}
```

⇓

```
void processCookieHeader(byte bytes[], int off, int len ) { ...
  if (bytes[startValue]=='1' && endValue==startValue+1 ) {
     version= getCookieHeaderVersion();
     if(log.isDebugEnabled()) log("Found version=1"); // aspect
  }  ...
}
int getCookieHeaderVersion() {   // extracted method
  return 1;
}
```

Fig. 4. Example of the application of Transformation 18 in the Tomcat system.

```
void log(Transaction tx, Date execTime, Turn myTurn) {
 ...
 try {                   /       / aspect
   myTurn.start();
   log.add(new TransactionTimestamp(tx, exeTime));
 }                   /           / aspect
 catch (Exception e) { ... }    // aspect
 finally { ... }                // aspect
 ...
}
```

⇓

```
void log(Transaction tx, Date execTime, Turn myTurn) {
 ... logHelper(tx, executionTime, myTurn);  ....
}
void logHelper(Transaction tx, Date execTime, Turn myTurn) {
  try {                   /      / aspect
    myTurn.start();
    log.add(new TransactionTimestamp(tx, execTime));
  }
 catch (Exception e) { ... }    // aspect
 finally { ... }                // aspect
}
```

Fig. 5. Example of the application of Transformation 21 in the Prevayler system.

```
void receiveTransactionFromServer() throws IOException,ClassNotFoundException {
  Object transactionCandidate= _fromServer.readObject();
  ....
  Date timestamp= (Date) _fromServer.readObject();
  _clock.advanceTo(timestamp);     // aspect
  ....
}
```

⇓

```
void receiveTransactionFromServer() throws IOException,ClassNotFoundException {
  Object transactionCandidate= _fromServer.readObject();
  ....
  Date timestamp= (Date) getTimeStamp();
  _clock.advanceTo(timestamp);      // aspect
  ....
}
Object getTimeStamp() throws IOException, ClassNotFoundException {
   return _fromServer.readObject();
}
```

Fig. 6. Example of the application of Transformation 23 in the Prevayler system.

Table 3 summarizes the transformations applied in the JAccounting system to modularize the concerns related to starting, committing and rollbacking transactions. As can be observed, a total of 60 transformations were required to enable extraction to aspects, including 30 statement reorderings and 30 ad hoc transformations. After their application, these transformations were responsible to enable 45 join points in the base code, i.e., on average 1.33 transformations were required per join point advised in the base system. As a consequence, the refactored system – after the application of the object-oriented transformations and the aspect-oriented refactorings – shows a high degree of dynamic crosscutting. Basically, this metric is defined as the radio between the number of join points by the number of pointcut descriptors declared in the aspects of a given system. The higher the ratio value, the greater the benefits generated by aspects in terms of eliminating duplicated and scattered code. In the refactored JAccounting, the degree of dynamic crosscutting is 11.25, i.e., the pointcuts of the aspect-oriented version of the system affect on average 11.25 join points of the base system.

Table 3
Transformations in the JAccounting system

JAccounting	
	Qty
Transformation 1	1
Transformation 5	14
Transformation 4	15
Ad hoc	30
Total (T)	60
Join points (P)	45
Pointcuts (C)	4
T/P	1.33
P/C	11.25

Table 4 summarizes the transformations applied in the JHotDraw, Prevayler and Tomcat systems. In JHotDraw, 18 transformations were needed, including 15 transformations in charge of untangling crosscutting code from expressions (Transformation 17). The Prevayler refactorization has demanded 8 method extraction transformations. In the Tomcat, 96 transformations were needed, including 86 method extractions (Transformations 18, 19, 21, 23, and 24).

Table 4
Transformations in the JHotDraw, Prevayler, and Tomcat systems

JHotDraw		Prevayler		Tomcat	
	Qty		Qty		Qty
Transformation 5	1	Transformation 19	1	Transformation 1	6
Transformation 16	1	Transformation 22	1	Transformation 2	3
Transformation 17	15	Transformation 23	2	Transformation 16	1
Transformation 18	1	Transformation 24	2	Transformation 18	5
Ad hoc	0	Ad hoc	2	Transformation 19	1
Total (T)	18	Total (T)	8	Transformation 21	1
Join points (P)	86	Join points (P)	55	Transformation 23	17
Pointcuts (C)	86	Pointcuts (C)	40	Transformation 24	62
T/P	0.21	T/P	0.15	Ad hoc	0
P/C	1	P/C	1.37	Total (T)	96
ITDs	56	ITDs	65	Join points (P)	258
				Pointcuts (C)	236
				T/P	0.37
				P/C	1.09
				ITDs	52

In the systems described in Table 4, the number of transformations per join point has ranged from 0.15 (in the Prevayler) to 0.37 (in the Tomcat). Such ratios are inferior to the ones found in the JAccounting refactorization. This observation is explained by the type of concern refactored. In the JAccounting, the target of the modularization was a homogenous concern, that is a concern that requires the application of the same advice in several parts of the system [9]. In this way, transformations were extensively employed to guarantee the levels of consistency expected by the pointcut descriptor language of AspectJ. As a result, it was possible to define pointcuts presenting high levels of dynamic crosscutting (superior to 11 join points per pointcut). On the other hand, in the JHotDraw, Prevayler, and Tomcat systems, the target of the aspectization were heterogeneous concerns, i.e., concerns that require the application of different code in different parts of the base application. For this reason, transformations were less demanded in such systems, since most of the aspects rely on intertype declarations (ITDs) to affect the base code. Consequently, such systems have presented low levels of dynamic crosscutting, ranging from around 1 (in the JHotDraw and Tomcat) to 1.37 (in the Prevayler).

3.1. Discussion

The following lessons were learned in the case studies described in this section:

- Object-oriented transformations are critical when aspectizing legacy systems. In our case study, the system that required the less amount of transformations was the Prevayler, where transformations were responsible to enable 15% of the join points advised by the aspects of the system. In the Prevayler and Tomcat systems, 21% and 37% of the join points were generated by transformations, respectively. In the JAccounting, due to ad hoc transformations, the number of transformations was 33% superior to the number of advised join points.
- Object-oriented transformations are required both in the modularization of homogeneous and heterogeneous concerns. However, they are more important when refactoring homogeneous concerns, since in this case it is fundamental that the concerns appear consistently in the base application. For example, in the modularization of transaction handling in the JAccounting system (a well-known homogeneous concern), a total of 60 transformations were required in order to capture 45 join points. It is also important to mention that homogenous concerns are the kind of concern that take more benefit from aspect-oriented implementations. The reason is that aspects are the right abstractions to modularize such concerns in a single module, eliminating the need of scattering and tangling implementations. Table 5 shows the number of lines of code for both the object-oriented (before applying any transformation) and aspect-oriented versions of the evaluated system. It can be observed that the aspectization of the JAccounting was the only one with a reduction in the number of lines of code. In the other systems, the number of lines has increased, due to the limited degree of dynamic crosscutting observed and to the extra syntax required to declare aspects and to apply transformations (particularly method extractions). The aspectization of the Prevayler has showed the greatest increase in lines of code (+44%), since it includes six concerns while in the other systems only one concern was modularized.

Table 5
Lines of code of the OO and AO versions of the evaluated systems

	LOC	
	OO version	AO version
JAccounting	11676	11477 (−1.7%)
JHotDraw	40022	40805 (+2%)
Prevayler	2418	3488 (+44%)
Tomcat	45107	45642 (+1.2%)

- It is very challenging to automate object-oriented transformations, even supposing that crosscutting concerns have been previously identified by an aspect mining tool. The main complexity is related to statement reordering and ad hoc transformations. As described in Section 2.1 (and in Appendix A), to evaluate if statements `A;B` can be reordered to `B;A` requires determining system-wide data and control dependencies between `A` and `B`. Moreover, the position of a statement in the code can be related to the logic of the concern being refactored. This is the case for example of logging, where reordering the calls to the logging API usually makes the logged messages incorrect or less accurate. An example can be seen in Fig. 4, where despite having no dependencies to other statements the `log` call cannot be moved to the beginning of the method body. Similarly, ad hoc transformations as suggested by their name are very dependent from the data and control flow of a particular application. For this reason, it is even complex to provide a step-by-step description of how to carry out an ad hoc transformation.

4. Related work

Related work can be arranged in four groups: object-oriented refactoring tools, aspect-oriented refactoring tools, catalogs of aspect-oriented refactorings and experiences in refactoring legacy applications.

4.1. Object-oriented refactoring tools

Because most refactorings require non-trivial manipulation of the codebase, program transformation systems can be used to automate refactoring application [27,31]. Early work in this area was done by Darlington and Burstall, concerning transformations to improve the performance of functional programs [5]. More recent systems include Stratego [30], TXL [11], and JunGL [29]. Stratego is a transformation system based on the paradigm of rewriting rules and strategies to control the process of transforming the program code. TXL is a another transformation language based on term rewriting and first order functional programming. JunGL is a domain-specific language that relies on predicates and path queries embedded within an ML-based language. As mentioned in Section 3, statement reordering transformations are usually more complex than common OO refactorings, since they depend on system-wide data and control flow information, which are not provided by the mentioned transformation systems.

4.2. Aspect-oriented refactoring tools

The study of the object-oriented transformations described in this paper has been motivated by an experience with the AOP-Migrator tool, an Eclipse plug-in that automates six refactorings commonly used to support migration from OOP to AOP [2,3]. The experience has showed that in most cases the supported refactorings cannot be directly applied, since they require preconditions that are usually not present in legacy code. This has motivated us to work on a catalog of enabling OO transformations and to investigate in details the use of such transformations in four Java applications, including two systems – JAccounting and JHotDraw – that have been aspectized with the support of AOP-Migrator. Liu, Batory, and Lengauer describe a refactoring methodology to decompose a monolithic program into a set of features. The methodology is supported by a tool that uses the AHEAD feature-oriented programming system to modularize features. Despite using another separation of concern technology, the authors also observed that transformations are needed when the code of two or more features is tangled in the same method. Anbalagan and Xie have proposed an automated approach for mining aspects and inferring pointcuts from legacy Java applications [1]. When the identified crosscutting concern is in the middle of a method, the tool that automates their approach relies on the statements before and after the concern to automatically infer a pointcut expression. However, as observed by the authors, it is possible that the statement before and after the concern may be repeated more than once in the method body or that such statements cannot be captured by the pointcut language of AspectJ. In such situations, the tool fails to infer the pointcuts since it does not include support to OO transformations. Concluding, the three mentioned tools reinforce the argument of Section 3 that although critical to the success of aspect extraction OO transformations cannot be fully automated.

4.3. Aspect-oriented refactoring catalogs

Monteiro and Fernandes have described a set of refactorings for extracting crosscutting concerns, restructuring the internals of aspects and dealing with inheritance hierarchies of aspects [23–25]. Another well-known catalog of aspect-oriented refactorings was proposed by Laddad [21]. Hannemann, Murphy, and Kiczales describe a role-based approach for aspect-oriented refactoring, where developers should map abstract roles to concrete program elements. Cole and Borba describe a set of aspect-oriented programming laws that are useful for deriving refactorings for AspectJ [8]. They recognize that the presented laws require subtle preconditions whose definition was the most challenging task of their research. As described in Section 1, neither of the previous catalogs are accompanied by an in-depth investigation about OO transformations.

4.4. Refactoring experiences

Colyer and Clement have described an experience on using AspectJ to refactor both homogeneous and heterogeneous concerns in a industrial middleware product-line platform [9]. Still in the middleware domain, Zhang and

Jacobsen have reported the use of AspectJ to modularize crosscutting concerns from a CORBA-based middleware [33]. Surprisingly, both works do not mention the need of OO transformations. More recently, Castor and colleagues have conducted a study about the aspectization of exception handling code [12]. Particularly, they report the difficulty in the modularization of `try-catch` blocks when they are tangled in the middle of methods (which demands the use of Transformation 21 described in this paper) and when the exception handler needs to access local variables from the base code (which demands the use of Transformation 24). 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 [32]. 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 reorderings and possibly ad hoc transformations (as was the case of transaction handling in JAccounting described in Sections 2 and 3).

5. Conclusions

In this paper, we have documented 24 transformations that can be applied to object-oriented code in order to enable join points that can be captured by the pointcut language supported by AspectJ. Basically, such transformations include statement reorderings and method extractions. We have also quantified the use of these transformations in four legacy Java systems that have been the target of aspect-oriented refactorings. This experience allowed us to conclude that OO transformations are critical in aspect-oriented refactoring processes, particularly when they involve the modularization of homogeneous concerns. Despite their importance, OO transformations cannot be easily automated, since they require complex, system-wide data and control dependency analysis.

In the future, we intend to work in new case studies. We also intend to design and implement a tool that supports mapping crosscutting concerns to code, similar to systems such as ConcernMapper [28] and Concern Manipulation Environment (CME) [7]. This tool should also indicate if the mapped crosscutting concerns can be extracted to aspects (without any transformation) or not. The idea is to provide information about the amount of transformation that must be applied to a system before starting its aspectization.

Acknowledgments

This research was supported by a grant from The State of Minas Gerais Research Support Foundation (FAPEMIG). We thank the anonymous referees for making several suggestions that improved the paper. We also thank Tarik Melo for reimplementing the logging concern of the Tomcat system using aspects.

Appendix A. Formal definition

This appendix provides a formal definition for the 24 transformations proposed in the paper. Our notation resembles the "push rules" proposed by Harman et al. to construct amorphous slices[6] [17]. Following their notation, an OO transformation is described by a rule of the form:

$$\frac{A}{B \Rightarrow C}$$

where A denotes the precondition required by the transformation and B and C are fragments of code. This rule should be interpreted in the following way: if A holds, then any code fragment that conforms to B can be transformed to C.

The provided definitions use the following helper functions (`st` denotes a block of statements):

- **def**(`st`): returns the set of variables that may be defined by the execution of `st` (including local variables, heap variables, and static variables).
- **ref**(`st`): returns the set of variables that may be referenced by the execution of `st`.
- **exit**(`st`): returns true if `st` includes a `return` or `throw` statement.
- **ret**(`st`): returns true if `st` includes a `return` statement.
- **call**(`m`, `st`): returns true if `st` includes a call to method (or constructor) `m`.
- **get**(`f`, `st`): returns true if `st` references the value of field `f` (`st` must be a single statement).
- **set**(`f`, `st`): returns true if `st` sets the value of field `f` (`st` must be a single statement).

In our definition, concatenation of two blocks of statements is represented by the juxtaposition of their descriptions. For example, **ret**($\mathtt{st}_1\ \mathtt{st}_2$) returns true if $\mathtt{st}_1$ or $\mathtt{st}_2$ includes a `return` statement. Moreover, `st[exp]` exposes the expression `exp` that is part of statement `st`. Finally, the definition of statement reordering transformations relies on the following function:

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

This function defines whether the order of statements $\mathtt{st}_1$ and $\mathtt{st}_2$ can be changed. This is possible when: $\mathtt{st}_1$ does not define any variable that is referenced by $\mathtt{st}_2$; $\mathtt{st}_1$ does not reference any variable defined by $\mathtt{st}_2$; both $\mathtt{st}_1$ and $\mathtt{st}_2$ do not define any variable in common;

[6] An amorphous slice is a variation of slicing with a more relaxed syntactic restriction. In the computation of traditional, syntax-preserving slices, only statement removals can be applied to the codebase; in amorphous slices, other transformations that reduce the size of the slice are allowed. In the work of Harman and colleagues, push rules are transformations designed to reduce inter-dependencies between statements, by pushing assignments forward in the code.

and $\mathtt{st}_2$ does not affect the execution of the program in a way that precludes the execution of $\mathtt{st}_1$ (for example, throwing an exception or executing a `return`).

Figs. A.1 and A.2 present the definition of the statement reordering transformations (T1–T17).

Fig. A.3 presents the definition of the method extraction transformations (T18–T24). The rules rely on a brace to delimit the code that must be extracted to a method:

$$\mathtt{st}\ \underbrace{st_1\ st_2\ \ldots st_n}_{\mathtt{m}}\ \mathtt{st}' \Rightarrow \mathtt{st\ m(p)\ st}'$$

(T1) $$\frac{\mathbf{chg}(\mathtt{st},\mathtt{cc})}{\mathtt{mtd}\,\{\mathtt{st\ cc\ st}'\} \Rightarrow \mathtt{mtd}\,\{\mathtt{cc\ st\ st}'\}}$$

(T2) $$\frac{\mathbf{chg}(\mathtt{cc},\mathtt{st})}{\mathtt{mtd}\,\{\mathtt{st}'\ \mathtt{cc\ st}\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}'\ \mathtt{st\ cc}\}}$$

(T3) $$\frac{\mathbf{chg}(\mathtt{cc},\mathtt{st}_2),\ !\mathbf{ret}(\mathtt{st}_1\,\mathtt{st}_2\,\mathtt{st}_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{return}\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{return}\ \mathtt{st}_3\}}$$

(T4) $$\frac{\mathtt{st}_3 = \mathtt{m(p)}\,\mathtt{st}_3',\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2),\ !\mathbf{call}(\mathtt{m},\mathtt{st}_1\,\mathtt{st}_2\ \mathtt{st}_3')}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T5) $$\frac{\mathtt{st}_2 = \mathtt{st}_2'\ \mathtt{m(p)},\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2),\ !\mathbf{call}(\mathtt{m},\mathtt{st}_1\,\mathtt{st}_2'\ \mathtt{st}_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T6) $$\frac{\mathtt{st}_3 = \mathtt{new\,T(p)}\,\mathtt{st}_3',\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2),\ !\mathbf{call}(\mathtt{new\,T},\mathtt{st}_1\,\mathtt{st}_2\ \mathtt{st}_3')}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T7) $$\frac{\mathtt{st}_2 = \mathtt{st}_2'\ \mathtt{new\,T(p)},\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2),\ !\mathbf{call}(\mathtt{new\,T},\mathtt{st}_1\,\mathtt{st}_2'\ \mathtt{st}_3)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T8) $$\frac{\mathbf{chg}(\mathtt{st},\mathtt{cc})}{\mathtt{catch(e)}\{\mathtt{st\ cc\ st}'\} \Rightarrow \mathtt{catch(e)}\{\mathtt{cc\ st\ st}'\}}$$

(T9) $$\frac{\mathbf{chg}(\mathtt{cc},\mathtt{st})}{\mathtt{catch(e)}\{\mathtt{st}'\ \mathtt{cc\ st}\} \Rightarrow \mathtt{catch(e)}\{\mathtt{st}'\ \mathtt{st\ cc}\}}$$

where `cc` are crosscutting statements; `mtd` is a method declarator; `m` is a method name; `p` is a list of actual parameters; `T` is a class name.

Fig. A.1. Transformations T1–T9.

(T10) $$\frac{\mathtt{st}_3 = \mathtt{st}_3'\,\mathtt{st}_3'',\ \mathbf{get}(\mathtt{f},\mathtt{st}_3'),\ !\mathbf{get}(\mathtt{f},\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{st}_3''),\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T11) $$\frac{\mathtt{st}_2 = \mathtt{st}_2'\,\mathtt{st}_2'',\ \mathbf{get}(\mathtt{f},\mathtt{st}_2''),\ !\mathbf{get}(\mathtt{f},\mathtt{st}_1\ \mathtt{st}_2'\ \mathtt{st}_3),\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T12) $$\frac{\mathtt{st}_3 = \mathtt{st}_3'\,\mathtt{st}_3'',\ \mathbf{set}(\mathtt{f},\mathtt{st}_3'),\ !\mathbf{set}(\mathtt{f},\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{st}_3''),\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T13) $$\frac{\mathtt{st}_2 = \mathtt{st}_2'\,\mathtt{st}_2'',\ \mathbf{set}(\mathtt{f},\mathtt{st}_2''),\ !\mathbf{set}(\mathtt{f},\mathtt{st}_1\ \mathtt{st}_2'\ \mathtt{st}_3),\ \mathbf{chg}(\mathtt{cc},\mathtt{st}_2)}{\mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{cc}\ \mathtt{st}_2\ \mathtt{st}_3\} \Rightarrow \mathtt{mtd}\,\{\mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}\ \mathtt{st}_3\}}$$

(T14) $$\frac{\mathbf{chg}(\mathtt{st},\mathtt{cc})}{\mathtt{static}\,\{\mathtt{st\ cc\ st}'\} \Rightarrow \mathtt{static}\,\{\mathtt{cc\ st\ st}'\}}$$

(T15) $$\frac{\mathbf{chg}(\mathtt{cc},\mathtt{st})}{\mathtt{static}\,\{\mathtt{st}'\ \mathtt{cc\ st}\} \Rightarrow \mathtt{static}\,\{\mathtt{st}'\ \mathtt{st\ cc}\}}$$

(T16) $$\frac{\mathbf{chg}(\mathtt{cc}_1,\mathtt{st}_2)}{\mathtt{st}_1\ \mathtt{cc}_1\ \mathtt{st}_2\ \mathtt{cc}_2\ \mathtt{st}_3 \Rightarrow \mathtt{st}_1\ \mathtt{st}_2\ \mathtt{cc}_1\ \mathtt{cc}_2\ \mathtt{st}_3}$$

$$\frac{\mathbf{chg}(\mathtt{st}_2,\mathtt{cc}_2)}{\mathtt{st}_1\ \mathtt{cc}_1\ \mathtt{st}_2\ \mathtt{cc}_2\ \mathtt{st}_3 \Rightarrow \mathtt{st}_1\ \mathtt{cc}_1\ \mathtt{cc}_2\ \mathtt{st}_2\ \mathtt{st}_3}$$

(T17) $$\frac{\mathtt{cc\ instanceof\ type(exp)}}{\mathtt{st[cc(exp)]} \Rightarrow \mathtt{t}_1\mathtt{=exp;\ t}_2\mathtt{=cc(t}_1\mathtt{);\ st[t}_2\mathtt{]}}$$

where `cc` are crosscutting statements; `mtd` is a method declarator; `f` is a eld name; `e` is an expression; $\mathtt{t}_1$ and $\mathtt{t}_2$ are local variables.

Fig. A.2. Transformations T10–T17.

(T18)

$$\mathtt{st_1}\ \underbrace{st}_{\mathtt{m}}\ \mathtt{cc\ st_2} \Rightarrow \mathtt{st_1\ m(p)\ cc\ st_2}$$

$$\mathtt{st}[\underbrace{exp}_{\mathtt{m}}]\ \mathtt{cc\ st'} \Rightarrow \mathtt{st[m(p)]\ cc\ st'}$$

(T19)

$$\mathtt{st_1\ cc}\ \underbrace{st}_{\mathtt{m}}\ \mathtt{st_2} \Rightarrow \mathtt{st_1\ cc\ m(p)\ st_2}$$

$$\mathtt{st'\ cc\ st}[\underbrace{exp}_{\mathtt{m}}] \Rightarrow \mathtt{st'\ cc\ st[m(p)]}$$

(T20)

$$\mathtt{st_1}\ \underbrace{st}_{\mathtt{m}}\ \mathtt{cc}\ \underbrace{st'}_{\mathtt{m}}\ \mathtt{st_2} \Rightarrow \mathtt{st_1\ cc\ m(p)\ st_2}$$

$$\mathtt{st_1}\ \underbrace{st}_{\mathtt{m}}\ \mathtt{cc}\ \underbrace{st'}_{\mathtt{m}}\ \mathtt{st_2} \Rightarrow \mathtt{st_1\ m(p)\ cc\ st_2}$$

(T21)

$$\mathtt{st_1}\ \underbrace{try\{st\}\ catch\{cc_1\}\ finally\{cc_2\}}_{\mathtt{m}}\ \mathtt{st_2} \Rightarrow \mathtt{st_1\ m(p)\ st_2}$$

(T22)

$$\mathtt{st_1}\ \underbrace{synchronized\{st\}}_{\mathtt{m}}\ \mathtt{st_2} \Rightarrow \mathtt{st_1\ m(p)\ st_2}$$

(T23)

$$\frac{\mathbf{call}(\mathtt{m(p)}, \mathtt{st_1\,st_2})}{\mathtt{mtd}\ \{\mathtt{st_1}\ \underbrace{m(p)}_{\mathtt{m'}}\ \mathtt{cc\ st_2}\} \Rightarrow \mathtt{mtd}\ \{\mathtt{st_1\ m'(p')\ cc\ st_2}\}}$$

$$\frac{\mathbf{call}(\mathtt{m(p)}, \mathtt{st_1\,st_2})}{\mathtt{mtd}\ \{\mathtt{st_1\ cc}\ \underbrace{m(p)}_{\mathtt{m'}}\ \mathtt{st_2}\} \Rightarrow \mathtt{mtd}\ \{\mathtt{st_1\ cc\ m'(p')\ st_2}\}}$$

(T24)

$$\mathtt{st_1\ v} = \underbrace{st}_{\mathtt{m}}\ \mathtt{st_2} \Rightarrow \mathtt{st_1\ v = m(p)\ st_2}$$

where `cc` are crosscutting statements; `mtd` is a method declarator; `m` or `m'` is the name of the extracted method; `v` is a local variable.
Blocks of code extracted to a method must have a single-entry, single-exit.

Fig. A.3. Transformations T18–T24.

This notation indicates that the transformation requires first the extraction of statements $\mathtt{st_1\ st_2 \ldots st_n}$ to a method named `m`. The extracted method can then be referenced in the right side of the rule. As usual in method extraction refactorings, the extracted code must be a single-entry, single-exit block of code.

References

[1] Prasanth Anbalagan, Tao Xie, Automated inference of pointcuts in aspect-oriented refactoring, in: 29th International Conference on Software Engineering (ICSE), May 2007.

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

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

[4] Gilad Bracha, William Cook, Mixin-based inheritance, in: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1990, pp. 303–311.

[5] R.M. Burstall, John Darlington, A transformation system for developing recursive programs, Journal of the ACM 24 (1) (1977) 44–67.

[6] Mariano Ceccato, Marius Marin, Kim Mens, Leon Moonen, Paolo Tonella, Tom Tourwé, A qualitative comparison of three aspect mining techniques, in: 13th International Workshop on Program Comprehension (IWPC), 2005, pp. 13–22.

[7] William Chung, William H. Harrison, Vincent J. Kruskal, Harold Ossher, Stanley M. Sutton Jr., Peri L. Tarr, Matthew Chapman, Andrew Clement, Helen Hawkins, Sian January, The concern manipulation environment, in: 27th International Conference on Software Engineering (ICSE), 2005, pp. 666–667.

[8] Leonardo Cole, Paulo Borba, Deriving refactorings for AspectJ, in: Fourth International Conference on Aspect-Oriented Software Development (AOSD), 2005, pp. 123–134.

[9] Adrian Colyer, Andrew Clement, Large-scale AOSD for middleware, in: Third International Conference on Aspect-Oriented Software Development, ACM Press, 2004, pp. 56–65.

[11] James R. Cordy, The TXL source transformation language, Science of Computer Programming 61 (3) (2006) 190–210.

[12] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhao, Alessandro Garcia, Cecilia Rubira, Exceptions and aspects: the devil is in the details, in: 14th International Symposium on Foundations of Software Engineering (FSE), 2006, pp. 152–162.

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

[14] Irum Godil, Hans-Arno Jacobsen, Horizontal decomposition of Prevayler, in: 15th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), 2005, pp. 83–100.

[16] Jan Hannemann, Gail C. Murphy, Gregor Kiczales, Role-based refactoring of crosscutting concerns, in: Fourth International Conference on Aspect-Oriented Software Development (AOSD), 2005, pp. 135–146.

[17] Mark Harman, Lin Hu, Malcolm Munro, Xingyuan Zhang, Dave Binkley, Sebastian Danicic, Mohammed Daoudi, Lahcen Ouarbya, Syntax-directed amorphous slicing, Automated Software Engineering 11 (1) (2004) 27–61.

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

[20] Gregor Kiczales, Jim Des Rivieres, Daniel G. Bobrow, The Art of the Metaobject Protocol, third ed., MIT Press, 1991.
[21] Ramnivas Laddad, Aspect-oriented refactoring, TheServerSide.com, 2003.
[22] Jia Liu, Don Batory, Christian Lengauer, Feature oriented refactoring of legacy applications, in: 28th International Conference on Software Engineering (ICSE), 2006, pp. 112–121.
[23] Miguel P. Monteiro, João M. Fernandes, Some thoughts on refactoring objects to aspects, in: VIII Jornadas de Ingeniera de Software y Bases de Datos (JISBD), 2003.
[24] Miguel P. Monteiro, João M. Fernandes, Towards a catalog of aspect-oriented refactorings, in: Fourth International Conference on Aspect-Oriented Software Development (AOSD), 2005, pp. 111–122.
[25] Miguel P. Monteiro, João M. Fernandes, Towards a catalogue of refactorings and code smells for AspectJ, Transactions on Aspect-Oriented Software Development 3880 (2006) 214–258.
[26] Gail C. Murphy, Albert Lai, Robert J. Walker, Martin P. Robillard, Separating features in source code: an exploratory study, in: 23rd International Conference on Software Engineering (ICSE), 2001, pp. 275–284.
[27] H. Partsch, R. Steinbrüggen, Program transformation systems, ACM Computing Surveys 15 (3) (1983) 199–236.
[28] Martin P. Robillard, Frdric Weigand-Warr, Concernmapper: simple view-based separation of scattered concerns, in: Eclipse Technology Exchange Workshop, ACM, 2005.
[29] Mathieu Verbaere, Ran Ettinger, Oege de Moor, JunGL: a scripting language for refactoring, in: 28th International Conference on Software Engineering (ICSE), 2006, pp. 172–181.
[30] Eelco Visser, Program transformation with Stratego/XT: rules, strategies, tools, and systems in StrategoXT-0.9, in: Domain-Specific Program Generation, Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, 2004, pp. 216–238.
[31] Eelco Visser, A survey of strategies in rule-based program transformation systems, Journal of Symbolic Computation 40 (1) (2005) 831–873.
[32] Isaac Yuen, Martin P. Robillard, Bridging the gap between aspect mining and refactoring, in: AOSD Workshop on Linking Aspect Technology and Evolution, 2007.
[33] Charles Zhang, Hans-Arno Jacobsen, Resolving feature convolution in middleware systems, in: 19th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, 2004, pp. 188–205.
[34] Charles Zhang, Hans-Arno Jacobsen, Efficiently mining crosscutting concerns through random walks, in: Sixth International Conference on Aspect-oriented Software Development (AOSD), 2007, pp. 226–238.