Sort-based Refactoring of Crosscutting Concerns to Aspects

Robin van der Rijst, Marius Marin, and Arie van Deursen

TUDelft

SE|RG

Sort-based Refactoring of Crosscutting Concerns to Aspects

Robin van der Rijst
Delft University of Technology
The Netherlands

rvdrijst@gmail.com

Marius Marin
Accenture
The Netherlands

Marius.Marin@accenture.com

Arie van Deursen
Delft University of Technology
The Netherlands

Arie.vanDeursen@tudelft.nl

ABSTRACT

Crosscutting concerns in object-oriented programming hinder evolution because of their symptoms: tangling and scattering. To benefit from the modularisation capabilities for crosscutting concerns provided by aspect-oriented programming (which prevent tangling and scattering) aspect-introducing refactoring can be used. The first step in aspect-introducing refactoring is identifying and documenting crosscutting concerns in existing code. The second step is refactoring the identified concerns to aspects.

This paper describes a tool called SAIR that can perform the second step of the aspect-introducing refactoring. For the first step, documenting, SAIR uses crosscutting concern *sorts*. Of the various possible sorts, SAIR currently supports the two most commonly encountered ones: Role Superimposition and Consistent Behavior. The refactoring towards aspects of concerns of these sorts is illustrated on an open source application (JHotDraw).

1. INTRODUCTION

The symptoms of crosscutting concerns (CCCs) in object-oriented systems, tangling and scattering, prevent easy software evolution [4]. Aspect-oriented software development (AOSD) provides a solution to these symptoms and their problems, by introducing constructs, such as aspects, that can be used to modularise CCCs [4].

AOSD can be applied to new programs, by incorporating aspects from the start, thereby preventing tangling and scattering from happening in the first place. However, in order to benefit from the modularisation capabilities of AOP in existing systems, aspects need to be introduced into these systems to refactor the implementation of crosscutting concerns.

Introducing aspects into existing systems by means of refactoring, is called *aspect-introducing refactoring*. Aspect-introducing refactoring consists of two steps: (1) the identification of crosscutting concerns in existing code and (2) the refactoring of those concerns to aspects, thereby introducing aspects to the (OO) system.

The first step of aspect-introducing refactoring is the process of lo-

cating crosscutting concerns in existing code, called *aspect mining* [5]. The results of aspect-mining indicate crosscutting concerns, which would ideally be modularised using aspects.

To document crosscutting concerns (identified by aspect-mining), we employ a concern documentation approach based on crosscutting concern *sorts* [6]. Concern sorts are aimed at providing a consistent solution to the documentation of crosscutting concerns, by organizing concerns based on specific implementation idioms. Each concern sort represents a class of concerns that share their idiom in a typical object-oriented implementation. Sorts are atomic, i.e. they cannot be divided in smaller concerns, and have associated a (desired) aspect-oriented modularization solution. Concrete occurrences of sorts in the code are called *sort instances*.

The documentation of concerns as sort instances in Java code is supported by a tool called SOQUET [8]. This tool allows the developer to document the sort instances as queries on the code. These queries indicate which code elements belong to the sort instance, and hence make up the crosscutting concern. The relation between these code elements is defined by the sort, and this relation determines how the sort instance should be refactored to aspects.

In our earlier work, we have used concern sorts as a starting point for refactoring crosscutting concerns in the JHotDraw application to aspects [7]. In this experiment, all refactorings were done manually, resulting in an aspect-oriented version of JHotDraw called AJHotDraw. In the present paper, we look into enabling tool support for these refactorings.

The remainder of this paper presents a proof-of-concept tool, SAIR, that performs the second step of aspect-introducing refactoring: the migration of the documented concerns to aspects. Section 2 introduces the main algorithm of SAIR. Section 3 and 4 present the sort specific algorithm details for two supported sorts: Role Superimposition and Consistent Behavior. Finally, Section 5 presents the results of a small refactoring experiment.

2. SAIR

SAIR[1], short for Sort-based Aspect Introducing Refactoring, is capable of migrating crosscutting concerns, expressed in terms of sort instances, to aspects. There are twelve CCC sorts [6], six of which can be documented using SOQUET. Currently, SAIR supports the refactoring of two common sorts: Role Superimposition and Consistent Behavior.

The algorithm behind SAIR can be described in two ways: with

---

[1]SAIR is available from `http://swerl.tudelft.nl/view/AMR/SAIR`

```
1  public class Example {
2      private int counter = 0;
3      public void someMethod(){
4          // ... [body] ...
5          counter++; // <= invisible in unprivileged aspect
6      }
7  }
```

Listing 1: Example of visibility problem

```
1  public interface CommandListener {
2      public void commandExecuted(EventObject e);
3      public void commandExecutable(EventObject e);
4      public void commandNotExecutable(EventObject e);
5  }
```

Listing 2: Example role-interface

a generic, sort-independent description and a sort-specific description. In this section, we will present the generic description and algorithm.

2.1 Input

The algorithm of SAIR expects a sort instance as input, and some input provided by the developer.

The sort instance indicates which elements of the code implement the crosscutting concern—these are the elements that will be migrated to an aspect. The relation between these elements is defined by the specific sort—this relation indicates *how* the elements should be migrated to an aspect. An example will be shown in the next section.

The developer needs to provide the target aspect. This can be a newly created aspect, or an existing aspect containing code. Additionally, depending on the sort being refactored, the user might be required to provide additional input.

2.2 Refactoring problems

Refactoring problems are problems that can occur during or after the refactoring. There can be compile errors after the refactoring, or certain elements of the concern cannot be migrated to the aspect.

SAIR tries to solve these refactoring problems by determining which problems will occur as soon as all input is available. For each of these problems, SAIR also determines a set of suggested solutions that will resolve the problem.

For each of the detected problems, the developer has to select one solution, thereby resolving the problem. Not all solutions may actually solve the problem—some solutions explicitly ignore or exclude part of the context, if the developer wants to manually solve the problem afterwards.

When all problems are resolved, SAIR applies all the selected solutions. This ensures the migration can finish without unexpected problems.

As an example of a simple problem that can be detected by the algorithm of SAIR, consider the code in Listing 1. If the body of someMethod() is moved to an unprivileged aspect during a refactoring, a compile error will occur because the body references the private field counter, which is invisible from the aspect.

Possible solutions that SAIR can currently suggest are making the aspect privileged, making the field public or creating and using getters and setters. Additionally, the problem can be ignored (resulting in a compile error) and fixed manually afterwards, or the refactoring of the body of someMethod() can be excluded.

2.3 Algorithm

The main steps of the generic algorithm are:

Start The sort is passed as a parameter to the algorithm.

Collecting input The input of the developer is collected. This is the target aspect and possibly sort dependent input.

Determining problems SAIR tries to determine the refactoring problems that will occur with the given sort instance and developer input. Possible solutions are also determined.

Resolving problems The developer selects one of the solutions for each of the refactoring problems, resolving the problems.

Applying solutions SAIR applies the selected solutions. This ensures the detected problems will not unexpectedly occur.

Migrate sort instance SAIR performs the migration of the sort instance to the target aspect. The concrete algorithm depends on the sort.

2.4 Implementation

SAIR is implemented as an Eclipse plug-in. It uses the Java Development Tools (JDT), AspectJ Development Tools (AJDT) and Language Toolkit (LTK). For sort instance input, SOQUET is used, which is also implemented as an Eclipse plug-in.[2]

In particular, SAIR extends SOQUET by adding a refactoring option to the context menu of sort instances. This option opens a LTK refactoring wizard, with the look and feel of ordinary Eclipse refactorings. The JDT and AJDT are used to perform code transformations.

An example of the input page of the LTK wizard of SAIR is shown in Figure 1.

3. ROLE SUPERIMPOSITION

One of the two sorts that is supported by SAIR, is the Role Superimposition (RSI) sort. This sort indicates the imposition of a secondary role on the primary role of classes. Sort instances of the RSI sort occur as a set of classes that implement a common *interface*.

An RSI sort instance consists of a set of methods in an interface that make up the role and the set of classes that implement the role, called the *imposees*. An example of a role interface for the CommandListener role is shown in Listing 2. Imposees with this role implement this interface and therefore have the secondary role of listening superimposed.

If the methods do not make up one complete interface, the role is called *virtual*. An example of a virtual role of an RSI sort instance is shown in Listing 3. This shows the Command interface, with a

[2] http://swerl.tudelft.nl/view/AMR/SoQueT
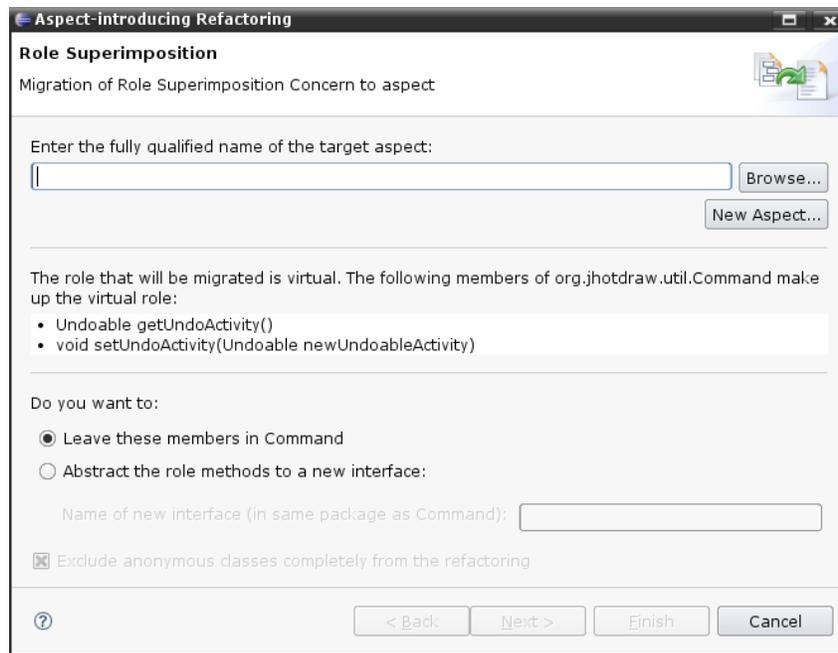
Figure 1: Providing input for virtual RSI refactoring

```
1  public interface Command {
2      Undoable getUndoActivity();        // undoable-role
3      void setUndoActivity(Undoable u);// undoable-role
4      Undoable createUndoable();         // undoable-role
5
6      void execute();                    // command-role
7      // ... [more command-role methods] ...
8  }
```

Listing 3: Example interface with virtual role

```
1  public class DeleteCommand implements Command {
2      public Undoable getUndoActivity(){
3          // ... [body] ...
4      }
5      public void setUndoActivity(Undoable u){
6          // ... [body] ...
7      }
8      public Undoable createUndoable() {
9          // ... [body] ...
10     }
11     // methods not related to undoable:
12     public void execute(){
13         // ... [body] ...
14     }
15     // [...]
16 }
```

Listing 4: Example `Undoable` imposee

virtual role of undoability. The undoable-role is virtual because the undo-related methods are part of a larger interface, that defines other Command-related methods.

A typical imposee of this virtual role is shown in Listing 4. This imposee *implements* the Command interface and because the virtual undoable role is defined in the Command interface, this imposee has the undoable role superimposed.

The desired aspect implementation of the RSI sort instance declares the role interface a parent of the imposees and implements the role methods as inter-type methods. When the role is virtual, declaring the complete interface a parent of the imposees imposes more than the virtual role. An option is to abstract the virtual role to a dedicated interface. This interface can then be declared parent of the original interface in the aspect[3].

The algorithm of SAIR for the RSI sort migrates the implementation to the aspect implementation. This RSI specific algorithm (the last

step in the generic algorithm) has the following main steps:

Non-virtual role For normal roles, the role interface is declared a parent of all the imposees in the aspect. The imposees no longer implement the interface directly (it is removed from the implements clause).

In the example of the role in Listing 2, there will be a declare parents: statement in the aspect for all imposees.

Virtual role For virtual roles, the developer can indicate whether virtual roles should be abstracted to a dedicated interface. If so, the new interface is declared a parent of the original interface.

In the example of Listing 3, the role of undoability will be abstracted to a new interface, e.g. UndoableCommand, which will be declared parent of the original Command interface.

Role members Finally, the implementation of the role methods is

---

[3]Ideally, the abstracted interface is declared a parent on the imposees directly, but in Java this will break method calls when the variable has the original interface as type, which will no longer have the method after abstraction.

```
1   public aspect UndoableCommandRole {
2       declare parents: Command extends UndoableCommand;
3
4       // implementation of undo-role for DeleteCommand:
5       public Undoable DeleteCommand.getUndoActivity() {
6           // ... [body] ...
7       }
8       public void DeleteCommand.setUndoActivity(
9                                       Undoable u) {
10          // ... [body] ...
11      }
12      public Undoable DeleteCommand.createUndoable() {
13          // ... [body] ...
14      }
15      // ... [implementation for other Commands] ...
16  }
```

Listing 5: Aspect implementation of virtual role superimposition

```
1   public class DeleteCommand implements Command {
2       public void execute(){
3           // ... [execute code] ...
4           setUndoActivity(createUndoable());
5       }
6       // ... [other Command methods] ...
7   }
```

Listing 6: Example Command with consistent behavior

moved from the imposees to the aspect as inter-type methods. In the example imposee of Listing 4, the undo-related methods will be moved to inter-type methods in the target aspect.

For the example virtual role of Listing 3 and imposee of Listing 4, the target aspect after refactoring will look like Listing 5.

4. CONSISTENT BEHAVIOR

The second sort supported by SAIR, is the Consistent Behavior (CB) sort. This sort indicates the consistent calling of a method from several points in the program. Sort instances of the CB sort therefore occur as a set of methods from which one method is called consistently.

A CB sort instance consists of the method that is consistently called (the consistent method) and the set of methods from which it is called, the context methods (with the consistent call). An example of a CB sort instance is shown in Listing 6, for the consistent setting of undo activities in the execute() method of Commands.

The desired aspect implementation of the CB sort instance consists of one or more pointcuts that capture the execution of the context methods. Advice is applied before or after the joinpoints captured by these pointcuts, and contains the consistent call.

The algorithm of SAIR for the CB sort migrates the implementation to the aspect implementation. This CB specific algorithm (the last step in the generic algorithm) has the following main steps:

Pointcut creation For each of the context methods, a pointcut is created in the target aspect that captures the execution of that pointcut. If the developer indicates that pointcut should be grouped, SAIR tries to create one pointcut for several context

```
1   public aspect UndoableCommandRole {
2       pointcut commandExecute(Command c) :
3           target(c) &&
4           (within(DeleteCommand) ||
5           // ... [other Commands] ...
6            within(PasteCommand)) &&
7           execution(public void execute());
8
9       after(Command c) : commandExecute(c){
10          c.setUndoActivity(c.createUndoable());
11      }
12  }
```

Listing 7: Aspect implementation of consistent behavior

methods if possible.
In the example of Listing 6, a pointcut will be created that captures the execute() method of DeleteCommand.

Advice creation For each of the created pointcuts, advice is created in the target aspect that will contain the consistent call. SAIR determines whether to use before or after advice based on the location of the consistent call in the context methods[4].
In the example of Listing 6, SAIR will create after advice for the pointcut that captures the execute method.

Call migration Finally, the consistent calls are removed from the context methods and placed in the corresponding advice. The aspect is now responsible for the consistent behaviour.

For the example in Listing 6, the target aspect after refactoring will look like Listing 7.

5. CASE STUDY

Using SAIR, we performed a small case study on JHOTDRAW 6.0b1. The SOQUET concern model used contained eight sort instances of RSI and CB sorts, concerning the Command implementation. The resulting aspects were compared with the manually created aspects of AJHOTDRAW [7].

Of the eight sort instances, seven were successfully refactored. The one that was not refactored, was a CB instance that consistently called a super constructor. This cannot be done from advice in AspectJ, so the call could not be migrated.

In all refactorings, SAIR detected some problems and provided satisfactory solutions to most. In two cases small manual refinements afterwards were needed, when the solutions did not provide the right result. For example, in Listing 8, the consistent call to get-AffectedFigures() is neither the first nor the last in the context, making it unclear to SAIR whether to use before or after advice.

A good solution would be to create before advice and move all the code preceding the consistent call to the advice as well, since it is directly related. However, SAIR does not (yet) provide this solution, so we told SAIR to use BEFORE advice explicitly and moved the related code to the advice manually after refactoring.

Not all Commands in JHOTDRAW were implemented in the same manner. This led to a situation where some context elements were

---

[4]If the call is not first or last in the method, a problem will have been detected in the problem step and the developer will have chosen how to handle the situation.

```
1   public void execute() {
2       // @SAIR: super.execute(); moved to advice ...
3       // @SAIR: setUndoActivi... moved to advice...
4       FigureEnumeration fe = view().selection();
5       List affected = CollectionsFactory.current()
6                                        .createList();
7       // ... [collecting affected figures] ...
8       fe = new FigureEnumerator(affected);
9       getUndoActivity().setAffectedFigures(fe);
10      deleteFigures(getUndoActivity().getAffectedFigures());
11      // @SAIR: view().checkDamage(); moved to advice ...
12  }
```

Listing 8: Consistent call in `DeleteCommand`

excluded from the refactoring because they could not be refactored. Refactoring them would require more pre- and postrefactoring than SAIR can handle with the current problem-solution framework.

Comparison with AJHOTDRAW showed that the automatically generated aspects are similar to the manually created aspects. One problem that was apparent, though, is that where SAIR always refactored one sort instance to one aspect, the implementation in AJHOTDRAW was sometimes spread over multiple aspects, leading to a cleaner implementation. This is an opportunity for improvement on SAIR.

Another difference is that the grouped pointcuts that are created by SAIR list all the classes to capture in an or-statement, as shown in Listing 7. The AJHOTDRAW pointcut is more advanced, capturing the complete hierarchy and only excluding those classes that should not be captured.

If SAIR should create the same structure, it would need to do a far more thorough analysis of the structure and hierarchy of the classes in question. For now, we see the creation of a better pointcut as an *aspect-oriented* refactoring, i.e. a refactoring that is concerned with refactoring only aspect code.

6. RELATED WORK

In this section, we summarize related research in the area of aspect-introducing refactoring.

Hannemann *et al.* have used the notion of *roles* being assigned to classes in design pattern implementations, to create a method called *role-based refactoring* [3]. These refactorings are limited to design-pattern related concerns.

A catalogue of aspect-oriented refactorings, among which are ten aspect-introducing refactorings, has been started by Monteiro and Fernandes [9]. The refactorings are related to structure of code, not to crosscutting concerns, although they can be used to build refactorings for crosscutting concerns.

Cole and Borba have described the use of *laws* that are guaranteed to preserve behaviour [2]. Similarly, Binkley *et al.* have introduced rules [1]. The granularity of these refactorings is very small and they are not directly related to crosscutting concerns, although they can be used to migrate them.

7. CONCLUSIONS

In this paper, we presented SAIR, a tool for automatic refactoring of crosscutting concerns to aspect-oriented programming. The tool

supports the refactoring of instances of two concern sorts: Consistent Behavior and Role Superimposition.

SAIR also implements a problem resolution mechanism that assists the user in dealing with the complexity of refactoring by providing a set of predefined solutions to common problems.

Future work will be carried out to identify additional problems that could occur in automatic refactoring to aspects, and to include solutions to these potential problems in the tool.

We also plan to implement refactoring support for all the sorts available in SOQUET, for a complete integration of the two tools.

8. REFERENCES

[1] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society.

[2] L. Cole and P. Borba. Deriving refactorings for aspectj. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 123–134, New York, NY, USA, 2005. ACM Press.

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

[4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[5] M. Marin, A. v. Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[6] M. Marin, L. Moonen, and A. van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM.

[7] M. Marin, L. Moonen, and A. van Deursen. An integrated crosscutting concern migration strategy and its application to JHotDraw. In *Proceedings of the IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 101–110. IEEE, 2007.

[8] M. Marin, L. Moonen, and A. van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 758–761, Washington, DC, USA, 2007. IEEE Computer Society.

[9] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, New York, NY, USA, 2005. ACM Press.

SERG