Sort-based Refactoring of Crosscutting Concerns to Aspects

Master’s Thesis

Robin J. van der Rijst
Sort-based Refactoring of Crosscutting Concerns to Aspects

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Robin J. van der Rijst
born in Hoorn, the Netherlands

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Abstract

Crosscutting concerns cannot be modularised in traditional object-oriented programming languages. They are therefore scattered throughout the system and tangled with the primary functionality of modules. As such, they hinder software evolution, the process of continuous adaptation of systems to changes in requirements and environments. Aspect-oriented programming allows the crosscutting concerns to be modularised in aspects. Existing systems can benefit from the introduction of aspects by aspect-introducing refactoring. Crosscutting concerns sorts are a way of consistently documenting crosscutting concerns in an existing system.

This thesis describes a refactoring algorithm and a proof-of-concept tool, SAIR that uses sorts as a basis for refactoring crosscutting concerns in existing systems to aspects. SAIR is able to refactor instances of two sorts: the Role Superimposition sort and the Consistent Behavior sort. A small case study is performed to assess the functionality of the algorithm and tool, and to identify opportunities for future work.
When I was looking for a master’s project, I happened to be a student assistant for Arie van Deursen, for the course on software quality and testing. When asked if he had any ideas, he introduced me to Marius Marin. Marius was working on an idea of his, called crosscutting concern sorts. His enthusiasm got me interested in using sorts for my master’s project. Thus the idea for this thesis was born.

I owe thanks to many people. First of all, I should thank Arie and Marius, for their continued support, thorough reviewing and proofreading of my thesis and their guidance in the course of my master’s project.

Of course, thanks go out to Leon Moonen and Koen Langendoen, who gave some of their valuable time so they could read this thesis and join the Thesis Committee.

Thanks also go to Max Jones. He generously granted permission to use one of his paintings on the cover of this thesis. I chose this painting because in a way it visualises abstractly the tangling of concerns in software systems. And I think it looks very nice on the cover of a thesis.

Last but not least, I like to thank the people closest to me. First of all my girlfriend, Sanne. Although very busy herself, she has always found the time to encourage me to go on, when I was losing motivation. And of course my parents, Wim and Esther, because it is with their help and support that I could make it to university in the first place.

I know I can never thank everyone who helped me personally. So let me summarise my gratitude to everyone with the following quote from the Twelfth Night1:

‘I can no other answer make but thanks
And thanks, and ever thanks’

William Shakespeare

Robin J. van der Rijst
Delft, the Netherlands
January 15, 2008

1I would further like to thank Google for providing me with this quote.
# Contents

Preface iii  
Contents v  

1 Introduction 1  
1.1 Software Evolution .......................... 1  
1.2 Problem: Crosscutting Concerns .............. 2  
1.3 Solution: Aspect-Oriented Programming ....... 2  
1.4 Status Quo .................................. 3  
1.5 Research Question ........................... 4  
1.6 Methodology ................................ 4  
1.7 Thesis Outline ............................... 5  

2 Background .................................. 7  
2.1 Advantages of Modularity ...................... 7  
2.2 Aspect-Oriented Programming ................. 9  
2.3 Refactoring .................................. 14  
2.4 Crosscutting Concern Sorts .................. 16  

3 Related Work ................................. 21  
3.1 Methods of Aspect-Introducing Refactoring .... 21  
3.2 Tool Support for Refactoring .................. 24  
3.3 Discussion of Limitations .................... 25  

4 Sort-based Aspect-Introducing Refactoring .... 27  
4.1 Introduction ................................ 27  
4.2 Introductory Example ......................... 27  
4.3 Generic Algorithm ........................... 30  
4.4 Implementation .............................. 38  
4.5 Usage ...................................... 47  
4.6 Summary ................................... 52  

v
Chapter 1

Introduction

To deal with complexity in current, large software systems, these systems can be decomposed into smaller parts. The abstraction mechanisms most programming languages use to realise such a decomposition are derived from the functional decomposition of a system [28]. The decomposition of a system in smaller parts, makes the system modular. Object-oriented programming languages introduce such modularity by using constructs like classes, methods and inheritance [42].

1.1 Software Evolution

Dealing with complexity is not only important for the initial development of a system. After the development of a system, it will still need to change. Bugs need to be fixed, modifications need to be made and extension need to be implemented. This means that as soon as a software system is developed, it will start to evolve.

The process of adapting and extending a software system to fix bugs that emerge, and cope with changing requirements and environments is called software evolution [31]. It is an inevitable process for any system that needs to stay functional under changing conditions and user demands.

High modularity eases software evolution, since more modular software is easier to understand [49, 56]. Changes to specific functionality are also localised in the relevant modules, making them more manageable than changes to systems with lower modularity. The use of object-oriented programming (OOP) in software development encourages a proper decomposition which provides a certain level of modularity and evolvability.

An important principle for software evolution is refactoring: the process of improving the program’s internal structure without altering the program’s behaviour [17, 46, 48]. Refactoring can be used to improve the quality of the code by making it more modular, readable, understandable and therefore, in general, easier to maintain, adapt and extend, thus evolve. Frequent refactoring helps to support the evolution of object-oriented (OO) systems.
1.2 Problem: Crosscutting Concerns

Although OOP is a popular programming paradigm, a problem with it is that it suffers from the tyranny of dominant decomposition [53]. This means that no matter how well the OO system is modularised, some functionality will not fit into the modularisation and is said to be crosscutting. These crosscutting concerns (CCCs) are typically tangled with the core functionality of the modules and scattered throughout the application, because they have, by definition, no single place in the decomposition [28].

A common example of a crosscutting concern is logging, where messages are written to a log file on well defined points in the execution of a program, such as the start of certain public methods or the occurrence of an exception. This concern is clearly scattered throughout an application and tangled with the primary functionality of the modules. Other examples include error handling, session management and persistence.

The implementations of design patterns also exhibit the symptoms of scattering and tangling, typical for CCCs. Design patterns are descriptions or templates that show how to solve typical problems, and can be used in many different situations [19]. These patterns often describe structural and functional relationships between classes or objects that cannot be captured by the decomposition. They can therefore be considered as templated implementations of CCCs.

The main problem with CCCs is that they hinder software evolution. They break the modular decomposition of the software system that would otherwise simplify evolution. On the one hand, modifications to the primary functionality of a module need to take into account the CCCs that crosscut that module. On the other hand, modifications to a single CCC implies the modification of many modules. Even after refactoring within the abilities of OOP—to improve modularity—, CCCs will still be tangled with the core functionality of the modules and scattered over different modules.

1.3 Solution: Aspect-Oriented Programming

The problems of CCCs in traditional programming paradigms would be solved if these concerns could also somehow be modularised. Such a modularisation would fix the symptoms of scattering and tangling, increasing modularity and simplifying software evolution.

This awareness has led to the development of a new programming paradigm: aspect-oriented programming (AOP)\(^1\) [28]. AOP extends existing paradigms such as OOP with constructs for the modularisation of CCCs. CCCs are encapsulated in modules called aspects. The possible benefits of AOP include higher modularity, better readability, greater separation of concerns and therefore programs that are easier to maintain and evolve [14, 30].

Aspects can be introduced to existing systems by refactoring CCCs to aspects. This will modularise CCCs that would otherwise hinder evolution, making the system aspect-oriented [6, 44]. Aspect-introducing refactoring is therefore an interesting approach for

---

\(^1\)Sometimes the term aspect oriented software development (AOSD) is used: software development using AOP as a programming paradigm.
solving the problem of CCCs in software evolution. It consists of two steps: first, CCCs need to be identified in existing code and second, these CCCs need to be moved to aspects.

1.4 Status Quo

The fact that introducing aspects to a system will improve modularity and ease software evolution, has already led to research in the identification of CCCs and the refactoring of identified CCCs to aspects.

1.4.1 Identifying crosscutting concerns

The process of identifying crosscutting concerns in non-aspect-oriented code is called aspect mining. There are a number of techniques and tools available that mine code for CCCs—for example using fan-in analysis (FINT) \[^2\] [41] or formal concept analysis (Dynamo) \[^3\] [54]. All these different techniques have their strengths and weaknesses. This suggests that a combination of the tools might improve the mining results [4, 5, 37].

The problem however, with combining the results of these techniques is that there is no consistent definition of a CCC. The characteristics of CCCs are known, but there is no consensus on what exactly one CCC is. This leads to different aspect mining techniques describing their results in different ways, making it hard to combine the results directly.

A solution to this inconsistency problem has been provided by Marin: crosscutting concern sorts [35, 36, 32]. A crosscutting concern sort is a generic description of a class of concerns. The concerns in these classes have common properties and are atomic: they cannot be subdivided into other CCCs. Concrete implementations of a CCC sort are called sort instances. Sort instances can be combined to form composite sorts. This makes it possible to express CCCs—including mining results—in terms of atomic sort instances.

Using sorts to document CCCs in code is supported by a tool called SoQUET\[^4\] [39]. Sort instances are documented using queries on the code. Each sort has a template that creates queries for the specific class of concerns. These queries are used to build a concern model: a tree structure that documents the CCCs in a system, with sort instances as leaves.

1.4.2 Refactoring CCCs to aspects

When refactoring the identified CCCs to actual aspects, the concerns have to be refactored from probably large, existing systems to aspects. If done manually, for non-trivial systems the task is not only labour intensive, but can easily introduce errors into the system. A (semi-)automated tool to aid the migration of CCCs to aspects is therefore a necessity. The need for proper aspect-introducing refactoring tools is recognised by other authors [8, 32].

There are a few techniques available in literature that describe aspect-introducing refactorings [2, 6, 15, 25, 29, 45, 55]. An overview of these techniques will be given in Sec-

\[^2\]http://swerl.tudelft.nl/bin/view/AMR/FINT
\[^3\]http://star.itc.it/dynamo/
\[^4\]http://swerl.tudelft.nl/bin/view/AMR/SoQueT
1.5 Research Question

The current situation can be summarised as follows. Aspect-oriented programming will make systems more modular by encapsulating CCCs, leading to easier software evolution. Aspects can be introduced into existing systems using aspect-introducing refactoring. Using sorts, CCCs in existing systems can be consistently identified and described. However, the few techniques available for the actual refactoring are not described in terms of sorts and lack consistency. Furthermore, aspect-introducing refactoring needs to be supported by tools but there are virtually no tools available.

Using this situation as a motivation, this thesis will provide an answer to the following research question:

How can crosscutting concern sorts be used for aspect-introducing refactoring?

1.6 Methodology

In order to answer the research question, this thesis will describe an algorithm for the refactoring of CCCs, expressed in terms of sort instances, from object-oriented code to aspects. This algorithm will be the basis of a proof-of-concept tool that will be able to migrate two sorts: the Role Superimposition sort and the Consistent Behavior sort [32]. Using this tool, we will perform a case study on the JHotDraw project [18]. Based on the results of this case study, we can evaluate how useful the algorithm and the tool are, and provide an answer to the research question.

The evaluation is based on two criteria: (1) the success rate of refactoring the concerns, and (2) the quality of the created aspects. By success rate we mean the percentage of all concerns that was successfully refactored. The quality of the created aspects is determined by the number of pointcuts and advice, the number of inter-type declarations, lines of code and the general readability of the aspect.
1.7 Thesis Outline

This thesis is structured as follows, in line with the methodology. First, in Chapter 2 we go into more depth on four relevant issues that were mentioned or assumed in this introduction with little explanation: the advantages of modularity, aspect-oriented programming, refactoring and sorts. In Chapter 3, related work on the refactoring of CCCs to aspects is presented and discussed. The algorithm behind, and the implementation of, the proof-of-concept tool for the sort-based refactoring of CCCs to aspects is presented in Chapter 4. The algorithms for refactoring two specific Sorts, namely the Role Superimposition sort and the Consistent Behavior sort, are discussed in Chapter 5 and Chapter 6. The results of applying the tool to an existing object-oriented system (JHOTDRAW) are presented in Chapter 7. Finally, the work presented in this thesis is evaluated in Chapter 8, where we discuss the contributions, provide a final answer to the research question and suggest future work based on the problems encountered and opportunities discovered.
Chapter 2

Background

This chapter introduces in more detail four issues that were mentioned or assumed in the introduction. These issues are key ideas and motivations behind this thesis. In the following chapters we will assume that the concepts described in this chapter are understood.

First, the implicitly assumed advantages of modularity are explored in Section 2.1. Next, in Section 2.2 the concepts of AOP are introduced, and we see how AOP can modularise CCCs using AspectJ as an example. In Section 2.3, OO refactoring and AO refactoring are presented. Finally, in Section 2.4 we show how sorts can solve the problem of granularity that CCCs have, and how the tool SOQUET can be used to document sorts in existing OO code.

2.1 Advantages of Modularity

It is often implicitly assumed that higher modularity leads to better code. This assumption is backed by an intuitive feeling that modularity leads to uncluttered code and that changes to specific functionality are localised. This section contains an example that supports this intuitive feeling.

2.1.1 Example of modularity advantage

Consider a simple Java [52] application that has a Display and several Figures that are displayed on this Display. Whenever the user makes changes to one of the Figures, these changes need to be reflected on the Display: the Figures need to notify the Display (as well as other interested classes) that there has been a change.

In object-oriented languages (like Java), the Observer pattern is typically used to create the required structure for this mechanism [19]. This pattern suggests two interfaces: Subject and Observer, shown in Listing 2.1. The Figure class will implement the Subject interface (as will other classes that need to be displayed on the Display). The Display will implement the Observer interface since it is observing changes in the Subjects. The code of the Figure will look like Listing 2.2.

The code in Listing 2.2 is simplified and not complete—coordinates alone are not enough for figures—but already the problem is apparent. The primary concern of the
2.1 Advantages of Modularity

Background

```java
public interface Subject{
    // register observer with subject for notification
    public void addObserver(Observer toAdd);
    // unregister observer from subject
    public void removeObserver(Observer toRemove);
}
public interface Observer {
    // notification that subject has changed
    public void notify(Subject changed);
}
```

Listing 2.1: The Subject and Observer interfaces

```java
public class Figure implements Subject{
    private Vector<Observer> observers = new Vector<Observer>();
    private int x = 0;
    private int y = 0;

    //method from Subject
    public void addObserver(Observer d){
        observers.addElement(d);
    }

    //method from Subject
    public void removeObserver(Observer d){
        observers.removeElement(d);
    }

    //method from Subject
    public void notifyObserver(){
        for (Observer ob : observers) {
            //notify is method of Observer, implemented by Display
            ob.notify(this);
        }
    }

    public void setX(int x){
        this.x = x;
        notifyObservers();
    }

    public void setY(int y){
        this.y = y;
        notifyObservers();
    }

    // ... [etc.] ...
}
```

Listing 2.2: Figure implementing Subject
Figures should be the representation of figures and their states (such as location). But the code in Listing 2.2 shows the secondary concern of maintaining a relation with Observers. The Figures keep a list of the Observers and notify them of every change. This clutters the class with extra methods, fields and calls.

If we could somehow modularise the notification functionality of the Figure class we would get an implementation as shown in Listing 2.3. The code related to the notification (the implementation of the Subject interface and the calls to the Observers) is removed from the class in Listing 2.3. This makes the class very clean and trivial to understand.

The fact that the notification call no longer appears in the methods does not mean notification doesn’t take place, but the notification concern is modularised. Another module (and not the Figure class) is responsible for the notification, and it should be the only functionality of that module. The modularisation makes it easier to understand the Figure class and other classes that would implement the Subject interface.

### 2.1.2 Limitations of OOP

The example clearly shows an improvement of code quality with increased modularity. The only problem with the example of Listing 2.3 is that OOP is incapable of such a modularisation. However, as we will show in the next section, aspect-oriented programming can modularise CCCs such as the notification concern in the Observer example.

### 2.2 Aspect-Oriented Programming

Aspect-oriented programming (AOP) extends traditional programming paradigms such as OOP\(^1\) by introducing aspects, which allow the developer to encapsulate CCCs. This section

\(^1\)AOP can also be applied to other paradigms (e.g. functional programming) but we look at OOP in this thesis since most research and development is done in the OOP area.
will first introduce the general properties all AOP languages have. Next, these properties will be explained using the AOP language AspectJ.

### 2.2.1 Properties of AOP languages

In general, AOP languages use a Join Point Model (JPM) that defines specific join points in the program’s execution where additional code can be executed. These join points indicate well-defined points in the execution of a program, such as the execution of a method, or the throwing of an exception. Join points are matched using pointcuts. Pointcuts are query-like constructs that will match only certain join points, e.g., only certain method calls. When pointcuts match a join point, code can be executed. This additional code is commonly called advice.

In addition, AOP languages may provide constructs to introduce extra members to classes and declare inheritance relationships. These constructs are commonly called inter-type declarations. All the AOP related constructs are grouped inside modules called aspects.

### 2.2.2 AspectJ as example

The most popular implementation of AOP is AspectJ, an extension of Java [27]. The syntax of AspectJ is increasingly being used as a description of AOP in general. We will use AspectJ to explain the properties of AOP in more detail.

To show how the features of AspectJ can be used, let us revisit the Observer example from Section 2.1. We have seen that the implementation of Figure in Listing 2.3 is an improvement over the implementation in Listing 2.2, but we have not seen how the notification mechanism is actually modularised. Using AspectJ, we can modularise the implementation as shown in Listing 2.4. This listing shows an aspect that is identified by the name FigureObserverProtocol and contains several typical constructs which will now be explained in a little more detail.

#### Inter-type declarations

AspectJ provides two types of inter-type declarations: those that add members to existing classes and those that add parents to existing classes. This way classes can be transparently extended with members, superclasses and interfaces for secondary functionality, without cluttering the class itself with those members and superclass and interface declarations.

The FigureObserverProtocol aspect contains inter-type parent declarations (lines 4-5) declaring Figure and Display to implement the Subject and Observer interfaces (see Listing 2.1), respectively. This makes those classes usable as if they directly implement the interface. Similarly, classes can be defined to extend other classes using the declare parents construct\(^2\).

The methods declared by the Subject and Observer interfaces need to be implemented by the Figure and Display classes. The inter-type method declarations (lines

\(^2\)Like in Java, multiple inheritance is still not possible and the AspectJ compiler will generate an error if declare parents will result in one class extending more than one other class
aspect FigureObserverProtocol{

    //inter-type parent declarations
    declare parents: Figure implements Subject;
    declare parents: Display implements Observer;

    //inter-type field declaration
    private Vector<Observer> Figure.observers = new Vector<Observer>();

    //inter-type method declarations
    public void Display.notify(Figure f) {
        this.update(f);
    }
    public void Figure.addObserver(Observer d) {
        observers.addElement(d);
    }
    public void Figure.removeObserver(Observer d) {
        observers.removeElement(d);
    }
    public void Figure.notifyObservers() {
        for (Observer ob : this.observers) {
            ob.notify(this);
        }
    }

    //pointcut definition
    pointcut changes(Figure f):
        target(f) &&
        call(void Figure.set*(..));

    //advice
    after(Figure f): changes(f) {
        f.notifyObservers();
    }
}
11-27) add the methods declared by the interfaces to these classes. These methods use an inter-type field declaration (line 8), which adds a field observers to the Figure class. This field is private which means only this aspect can access it.

**Pointcuts**

Pointcuts identify a set of join points where advice can be applied. AspectJ supplies a number of primitive pointcuts. New pointcuts can be built from other pointcuts and they can be named. To expose the context of the join points captured by the pointcut, pointcuts can have parameters that will be bound to a join point.

Our example aspect has one named pointcut called changes (lines 30-32). A Figure is passed as an argument and is used to expose the context. The actual matching is based on the primitive pointcuts target and call: the pointcut only matches if the joinpoint is an operation on the given Figure f, the target, and that operation is a call to a method of Figure starting with set and any number and type of arguments (...). Using the parameter Figure in the target pointcut exposes the matching figure to the user of the pointcut, which can be advice or another pointcut.

There are many more primitive pointcuts, and the JPM of AspectJ is much more powerful than this simple example. For a complete description of all these pointcuts, see [27].

**Advice**

Advice can be used to apply code when pointcuts are matched. The example uses the named pointcut changes to apply code after the pointcut has matched (lines 35-37). If the pointcut matches, it is executed first, and only after execution is the advice executed. The Figure f is the context exposed by the pointcut and can be used in the advice: f is the Figure matched as target in the pointcut changes.

This advice is after advice, but there are several other types of advice. The advice types after() returning and after() throwing only apply the advice after the pointcut is finished either normally or by throwing an exception, respectively. The before advice is executed before the matched pointcut continues. The around is executed instead of the pointcut. The pointcut can optionally be continued inside the advice body by calling proceed. This advice should return a value that is of the same type as the value that would be returned by the captured join points.

**How this example operates**

The example aspect modularises the notification concern, and operates as follows. When we call f.addObserver(d) to register a Display d to a Figure f as observer, the Display is added to the Vector observers declared for Figure on line 8. Next, when we call, for instance, f.setX(10) on Figure f, the pointcut changes (lines 30-32) is matched, since we call a method starting with set on an instance of class Figure. This means that the advice on lines 35-37 is executed after the method setX(10) returns and the inter-type method notifyObservers() of Figure f is called, which in turn calls the inter-type method notify(f) of Observer for all registered Observers.
Other AspectJ constructs

The constructs of AspectJ are not limited to those mentioned in this brief example. There are for instance many more primitive pointcuts than the call and target pointcuts used in the FigureObserverProtocol[27].

Some other constructs of AspectJ are concerned with error handling. Using the declare error and declare warning, specific join points can be declared to generate compiler errors or warnings if matched by a pointcut. For example, a compile error can be generated if a method is called from outside a set of packages in a program.

Using declare soft, checked exceptions thrown by specific join points can be wrapped in a SoftException. Checked exceptions in Java are non-runtime exception and when a method throws or re-throws a checked exception, it needs to declare this using a throws clause. Since SoftException is a runtime exception, wrapping checked exceptions in SoftExceptions, removes the need for a throws clause.

Another AspectJ feature is that of privileged aspects. Normally, aspects cannot access private members, or protected or package-visible members in other packages, from within inter-type methods and advice. If an aspect is declared privileged however, it can access private members and protected and package visible members from classes in other packages.

Aspects in AspectJ can also have their own members (methods, fields, inner classes, etc.). However, since aspects are not instantiated like classes, these members can be used by the aspect (in advice for example), but not by the inter-type declarations such as the methods in our example. Access from these methods would require the aspect member field to be static, which in our example means that all Figures would share the same (static) list of Observers.

When structural components of several aspects are the same and related, these components (advice, pointcuts, etc.) can be placed in an abstract aspect which is extended by the related aspects. This inheritance of aspects allows for even better modularisation. An AspectJ implementation of the Observer pattern that uses inheritance is described in more detail in [24].

A more complete description of the syntax and possibilities of AspectJ can be found in [27], where Kiczales et al. discuss AspectJ in detail. More information on the current developments of AspectJ is available from [1].

2.2.3 Research in Modularisation Benefits

To support the intuitive feeling of improvement based on examples like the above, some research has been conducted in the field of quality improvements by better modularity and the use of AOP. We briefly discuss these studies here.

Binkley et al. [2] studied quality improvement by the insertion of aspects, based on comparing values from software metrics before and after the refactoring. They do find an improvement in code quality based on their metrics, but they acknowledge the limited applicability of their results and suggest further research.
Guyomarc’h and Guéhéneuc [21] announce the development of AOP metrics based on OOP metrics. They expect that AOP influences metrics such as coupling, cohesion and depth of inheritance. In his Masters Thesis, [20] (French, English abstract) Guyomarc’h presents the results of applying the modified metrics and he concludes, based on a case study, that refactoring OO systems to AO systems (increasing modularity) will improve the quality of the system.

Rønningen and Steinmoen have investigated the improvement of readability through the use of AOP [49]. They argue that improved readability depends (also) on reduced complexity. AOP reduces complexity by dividing systems into smaller parts (modularisation) and therefore, according to their initial study on one system, increases readability.

In his Ph.D. thesis, Van Belle links software evolution to biological evolution and he shows that an improvement in modularity will lead to and improvement in evolvability [56]. He concludes with the suggestion that AOP can result in better evolvability due to the modularisation of CCCs.

Dhillon discusses software quality in [9] (Chapter 9). He identifies three categories of Software Quality Factors: Product operation factors, Product revision factors and Product transition factors. The latter two categories indicate software evolutionary activities such as maintenance. Five of the six quality factors that fall in these two categories are related to modularity. They indicates that increased modularity is an important factor in software evolution—and one of the goals of AOP.

### 2.3 Refactoring

Refactoring has the goal of improving the structure of the code. This means adapting the structure without changing the behaviour of the program.

Traditional OO refactorings improve the structure by making it as object-oriented as possible [46, 17]. These refactorings include extract interface, extract method, extract superclass and rename method; these names alone give a good insight in the goal of refactoring. A more detailed description of these and about ninety other refactoring can be found on the Refactoring Home website, maintained by Fowler [47].

These refactorings however, are limited by the capabilities of the OOP paradigm. For instance, one might want to make logging, persistence and even design pattern implementations modular and explicit somehow, instead of scattered and tangled. This cannot be done with OOP alone, because of the constraints OOP has regarding such CCCs. Refactoring CCCs to aspects might solve those problems by providing constructs for increased modularisation.

#### 2.3.1 Aspect refactoring

The term ‘aspect refactoring’ refers to refactoring in an aspect-oriented setting, but the exact meaning is ambiguous, judging from the different interpretations found in literature.

Let us define aspect code as the code located in the aspects and base code as the OO code without the aspects. Using this definition, aspect refactoring can in general refer to three types of refactoring [22, 60]:

---

Guyomarc’h and Guéhéneuc [21] announce the development of AOP metrics based on OOP metrics. They expect that AOP influences metrics such as coupling, cohesion and depth of inheritance. In his Masters Thesis, [20] (French, English abstract) Guyomarc’h presents the results of applying the modified metrics and he concludes, based on a case study, that refactoring OO systems to AO systems (increasing modularity) will improve the quality of the system.

Rønningen and Steinmoen have investigated the improvement of readability through the use of AOP [49]. They argue that improved readability depends (also) on reduced complexity. AOP reduces complexity by dividing systems into smaller parts (modularisation) and therefore, according to their initial study on one system, increases readability.

In his Ph.D. thesis, Van Belle links software evolution to biological evolution and he shows that an improvement in modularity will lead to and improvement in evolvability [56]. He concludes with the suggestion that AOP can result in better evolvability due to the modularisation of CCCs.

Dhillon discusses software quality in [9] (Chapter 9). He identifies three categories of Software Quality Factors: Product operation factors, Product revision factors and Product transition factors. The latter two categories indicate software evolutionary activities such as maintenance. Five of the six quality factors that fall in these two categories are related to modularity. They indicates that increased modularity is an important factor in software evolution—and one of the goals of AOP.

### 2.3 Refactoring

Refactoring has the goal of improving the structure of the code. This means adapting the structure without changing the behaviour of the program.

Traditional OO refactorings improve the structure by making it as object-oriented as possible [46, 17]. These refactorings include extract interface, extract method, extract superclass and rename method; these names alone give a good insight in the goal of refactoring. A more detailed description of these and about ninety other refactoring can be found on the Refactoring Home website, maintained by Fowler [47].

These refactorings however, are limited by the capabilities of the OOP paradigm. For instance, one might want to make logging, persistence and even design pattern implementations modular and explicit somehow, instead of scattered and tangled. This cannot be done with OOP alone, because of the constraints OOP has regarding such CCCs. Refactoring CCCs to aspects might solve those problems by providing constructs for increased modularisation.

#### 2.3.1 Aspect refactoring

The term ‘aspect refactoring’ refers to refactoring in an aspect-oriented setting, but the exact meaning is ambiguous, judging from the different interpretations found in literature.

Let us define aspect code as the code located in the aspects and base code as the OO code without the aspects. Using this definition, aspect refactoring can in general refer to three types of refactoring [22, 60]:

---
### 2.3 Refactoring

**Aspect-aware refactoring** – When refactorings are applied to base code of an aspect-oriented application, these refactorings have to take the aspects into account. For instance, when a *Rename Method* refactoring is performed, pointcut definitions may need to be updated as well. These OO refactorings need to be *aspect-aware* in order to change the aspect code as required. Aspect-aware refactorings are discussed in more detail in [22, 26, 50, 60].

**Aspect code refactoring** – With aspects come specific refactorings that refactor the aspect code only. Examples are *Extract Pointcut Definition* and *Extract Abstract Aspect*, which extract common pointcut definitions from advice into a named pointcut, and common structure into an abstract superaspect, respectively. Aspect code refactorings are discussed in more detail in [16, 43].

**Base-to-aspect code refactoring** – When CCCs are extracted from OO code into aspects, we call this base code to aspect code refactoring. To find CCCs that might be refactored to aspects, a process called *aspect mining* is used.

Some authors do not distinguish between aspect code refactoring and base-to-aspect code refactoring, grouping them as ‘AOP’ refactorings [26, 22]. We make the distinction because conceptually the creation of aspects from scattered and tangled CCCs is different from tidying up existing aspects.

Wloka identifies a fourth type of refactoring, which he calls aspect-to-base code refactoring [60]. Wloka argues that sometimes, after changing and refactoring a system, aspects only apply to one class and can therefore be refactored back to that class, removing the aspect. However, since the aspect represents a secondary concern for that class, it makes perfect sense to keep that concern in an aspect, and therefore these refactorings are not considered in the classification above.

#### 2.3.2 Aspect-introducing refactoring

The third type of aspect refactoring in Section 2.3.1, base-to-aspect code refactoring, is the type of interest to this thesis. This type of refactoring introduces new aspects by moving CCCs from object-oriented code to aspects. It is therefore referred to in this thesis as *aspect-introducing refactoring*. Aspect-introducing refactoring is a two-step process: first, the CCCs need to be identified and second, the identified CCCs need to be moved to aspects.

**Step 1: Aspect mining**

*Aspect mining* is the process of identifying crosscutting concerns in non-aspect-oriented code. There are several ways to mine existing code, looking for aspect opportunities. Code elements that may indicate the presence of CCCs are called *candidate seeds* [33]. Using manual analysis by a human expert, we can separate the proper seeds (*confirmed seeds*) from the *false positives*. Proper seeds that were not identified by the aspect mining technique are referred to as *false negatives*. A distinction can be made between *generative* and *explorative* (also called *query-based*) mining techniques [33].
Generative mining  Generative mining techniques aim at automatically generating CCC seeds of high quality with a low percentage of false positives and false negatives. Many generative approaches look for symptoms of scattering and tangling of code using program analysis techniques and identify code elements with these symptoms as candidate seeds.

Explorative mining  Explorative mining techniques rely on search patterns provided by the user. These input seeds are often created using generative techniques. Using these seeds as queries, tools support software navigation to identify the concerns related to these seeds.

Each technique has its strengths and weaknesses and a combination of different aspect mining techniques has been suggested to generate more, and more accurate results [4, 37]. The problem that was already mentioned in the introduction, is that these aspect mining techniques do not describe their results in a consistent way. The lack of a consistent definition of what exactly one CCC is, makes it very hard to combine the results of different techniques. The solution can be found in CCC sorts, described in Section 2.4.

Step 2: Migrating concerns to aspects

This step migrates the CCCs found in step 1 to aspects. It uses refactoring techniques to modularise the CCCs in an aspect, without changing the behaviour of the program. In line with the research question in Section 1.5, the goal of this thesis is to show how such a refactoring technique can be based on crosscutting concern sorts (see Section ).

A few refactoring techniques have been described in literature and they will be presented in Chapter 3. There, for each technique the drawbacks will be given. The main drawback of those techniques has to do with granularity. Some techniques describe refactoring of composite CCCs, tackling more than one CCC in one refactoring. Other techniques describe refactoring in very small steps that are oblivious to the actual CCC they are refactoring.

This granularity inconsistency of the refactoring techniques is also the result of a lack of definition for CCCs. The refactoring is not consistently expressed in CCCs, which makes it hard to directly feed the results of the aspect mining step into the refactoring. It also makes it hard to have tools that aid the refactoring, because the correlation between the code elements and the refactoring steps depends on the level of granularity in which the refactoring is expressed.

2.4  Crosscutting Concern Sorts

As mentioned in the introduction, CCCs tend to be described using inconsistent definitions. This leads to problems when the results of aspect mining techniques need to be combined and when these results need to be automatically refactored. Marin introduces a way of consistently describing the CCCs in code using sorts [35, 36, 32]. A crosscutting concern sort is a generic description of a class of concerns. These concerns are atomic and share three properties (from [32]):
2.4 Crosscutting Concern Sorts

- an intent (behavioural, design or policy requirements);
- a specific implementation idiom in an (object-oriented) language; and
- a (desired) aspect language mechanism that supports the modularisation of the sort’s concrete instances.

Concrete implementations of a CCC sort are called sort instances. The fact that sorts are atomic is an important property. It means that a sort instance is the smallest unit in which CCCs can be expressed. This means that all crosscutting concerns can be expressed in terms of sort instances. A concrete crosscutting concern can be built from one or more sort instances.

This thesis describes a proof-of-concept tool that support the automated refactoring of two specific sorts: Role Superimposition and Consistent Behavior. A list of thirteen sorts of crosscuttingness, including RSI and CB, with short explanations for each of the properties, can be found in [36].

2.4.1 Role Superimposition sort

Role Superimposition (RSI) is a sort that indicates that certain classes implement a certain role that indicates a secondary concerns. This role is usually an interface. Examples are implementations of design patterns like Observer, Command and Visitor, where classes get secondary roles related to the pattern but not to their primary concern.

The aspect implementation of the RSI sort will use inter-type declarations to implement the role in an aspect. The role is declared parent of the classes it is superimposed on using declare parents constructs. The role will be implemented using inter-type methods, which means the implementation of the role is inside the aspect instead of the classes that have the role superimposed.

2.4.2 Consistent Behavior sort

Consistent Behavior\(^3\) (CB) is a sort that indicates that a certain method is called consistently from specific contexts. This can indicate that these consistently called methods are not really part of the primary concern of the method. Examples are the notification of observers in the Observer pattern, or the checking of preconditions at the start of a method.

The aspect implementation of the CB sort will use pointcuts and advice to implement the consistent behaviour in an aspect. Pointcuts are used to match the contexts in which the consistent call is made. Advice is used to execute the consistent call before or after the contexts matched by the pointcuts.

2.4.3 SOQUET

SOQUET is a tool for consistently describing and documenting CCCs as sort instances [51]. It relies on queries to document crosscutting relations in code. Crosscutting concern sorts

\(^3\)We use the American English spelling of behavior only when we talk about the sort (since Marin emphet al. introduced sorts with this spelling [35]) and the British English spelling of behaviour otherwise.
2.4 Crosscutting Concern Sorts

Figure 2.1: Example of the Concern Model view in SOQUET

are used to provide query templates that indicate the relation of elements captured by the query.

SOQUET can be used for two main purposes. The first is that of explicitly documenting crosscutting concerns using the sort-based query system. This can be the result of the aspect mining step. The results of this documenting activity is stored in a concern model. This concern model can be used to build a tree structure of the CCCs in a system, with the sort instances—expressed as queries on the code—as leaves.

The second purpose of SOQUET is that of exploring the software using a previously build concern model. This model can be used to provide a deeper understanding of the code and the CCCs that are tangled and scattered throughout the system.

This thesis will append a third purpose to SOQUET, namely that of starting point of refactoring. The concern model is described consistently in terms of atomic crosscutting concern sorts and already contains queries that indicate which parts of the code correspond to which sort. This information provides the perfect basis for the refactoring step.

Example of Concern Model

An example of a concern model in SOQUET is shown in Figure 2.1. This figure shows a screenshot of the Concern Model view of SOQUET, containing the concern model for the ‘Command’ structure of the JHotDRAW source code.

The tree structure of the concern model can clearly be seen. There are two types of nodes: concern nodes and sort instance nodes. Concern nodes can contain other contain other concern nodes and sort instance nodes, while sort instance nodes cannot contain other nodes. The figure shows concern nodes as expandable bullets, while the sort instance nodes are shown as non-expandable open folders.

The root of the tree is a concern node named ‘JHotDraw’, indicating this is a concern model for JHotDRAW. This concern model only contains the ‘Commands’ concern, so that is the only child of the root. The ‘Commands’ node however, contains both concern nodes
and sort instance nodes. This means some concerns can be grouped further, for instance in the ‘UndoSupport’ concern.

Sort instances have a name and a description. The name is given by the developer who built the concern model, e.g. ‘PreExecutionCheck-ViewNotNull’ in the figure. The description is the part in square brackets after the name. It shows the type of sort (e.g. CB) and the context elements and their relation (e.g. methods in the ‘hierarchy of Command’ that ‘invoke’ the method ‘execute’).
Chapter 3

Related Work

This chapter will present previous efforts in the area of aspect-introducing refactoring. The chapter is organised in three parts. First, the known methods of refactoring of CCCs to aspects will be presented. Next, the tool support described in literature will be discussed. Finally, the main limitations of the methods and tools will be highlighted to emphasise the need for a proper tool supported aspect-introducing refactoring technique, as presented in this thesis.

3.1 Methods of Aspect-Introducing Refactoring

Methods of aspect-introducing refactoring are descriptions of how aspects can be introduced in existing object-oriented applications by migrating CCCs to new aspects. This section introduces all methods found in literature. The methods are arranged based on their granularity, from coarse to fine-grained.

3.1.1 Design-patterns and roles

Most design patterns have an aspect-oriented equivalent that generally increases the quality of the code. Hannemann and Kiczales identify aspect-oriented versions for most of the pattern and they show how code quality increases [24]. The implementations they suggest remove the roles from the participating classes and move them to aspects. Another improvement of the aspect-oriented solution is that, in many patterns, the core implementation can be abstracted and reused for several instances of the same pattern throughout the code.

Based on the notion of roles being assigned to different classes in design patterns, Hannemann et al. have devised a refactoring method they call role-based refactoring in [25, 23]. Their method uses a library of CCC refactorings where each CCC and the refactoring is defined abstractly in terms of roles. When a developer refactors he is required to map the roles of the CCC to program elements such as methods or classes. The abstract CCC defines the relationships between these elements. The refactoring then describes how to manipulate the mapped elements and what new aspect code to create. This way, the same abstract CCC refactoring can be used to refactor several instances of the same pattern, only with a different mapping of elements.
3.1 Methods of Aspect-Introducing Refactoring

Granularity

Refactorings based on design patterns and roles of elements therein, are coarse. The design patterns often cover more than one CCC, such as the imposing of a role as well as the enforcement of consistent behaviour. Furthermore, the refactorings associated with these design patterns are possibly big, touching large areas of the code—all code associated with a design pattern implementation—in a single refactoring.

3.1.2 Catalogue based

Monteiro and Fernandes present the start of a catalogue of AO refactorings [45]. They argue that a similar catalogue of object-oriented refactoring by Fowler et al. has been very influential as a basis for developers to rely on [17].

Code smells for AOP should indicate problems that can be improved by using proper AOP, i.e. by making optimal use of the AOP constructs. Some object-oriented code smells can be adapted for use as CCC indicators, but some new code smells are for aspects specifically, e.g. the aspect laziness code smell. This smell indicates that an aspect uses more inter-type declarations than necessary, in essence passing responsibility back to the original class. This is an example of an aspect-specific refactoring, since it concerns the aspect code, not base code, but the catalogue also defines aspect-introducing refactorings. These refactorings are grouped under the name of Refactorings for Extracting Features to Aspects in the catalogue.

Concretely, the catalogue gives ten aspect-introducing refactorings. However, simply applying these refactorings will yield less-than-perfect aspects. Therefore, the catalogue also contains seventeen refactorings to tidy up the new and existing aspects.

The format of the refactoring descriptions is similar to the one used by Fowler [17] and includes (1) a name, (2) a typical situation, (3) a recommended action, (4) a motivation stating the situation when applying the refactoring is desirable, (5) a detailed mechanics section where the actual work is described and (6) code examples.

Monteiro and Fernandes are not the first to present a list of concrete refactorings. Happenberg et al. describe three refactorings to extract CCCs from object-oriented code to aspects [22]. These refactorings are also included in the catalogue of Monteiro and Fernandes.

A collection of AO refactorings is also discussed by Laddad. His collection of refactoring is presented in the article he wrote on TheServerSide.com [29]. In this article he prescribes guidelines to ensure the refactorings are applied in a safe way and accurate pointcut definitions are created. The refactorings vary in scope from general (e.g. Extract Interface Implementation) to more concern specific (e.g. Extract Contract Enforcement).

However, the descriptions of Laddad’s refactorings lack a coherent structure like the one used by Fowler and Monteiro and sometimes consist of nothing more than a name and motivation. This makes them less directly applicable to source code and requires more from the developer. Monteiro and Fernandes also argue that the less general refactorings of Laddad can be decomposed into lower-level refactoring steps [45]. Nevertheless Laddad has interesting refactorings that are not present in the catalogue of Monteiro and Fernandes.
Finally, Tonella and Ceccato describe a list of refactorings based on the assumption that interfaces are often related to CCCs [55]. Monteiro and Fernandes call this the Interface Implementation code smell and provide refactorings that achieve the same extraction of interface-to-aspect in case the interface can indeed be regarded as a CCC.

Granularity

Refactorings from catalogues like Monteiro and Fernandes’ are less coarse than the refactorings based on design patterns and roles. These catalogue-based refactorings commonly apply to single elements of code, e.g. Move Field from Class to Inter-type and Replace Implement with Declare Parents. Although they can successfully be used to migrate CCCs to aspects, several refactorings need to be combined to completely migrate one CCC.

3.1.3 Program Slicing

The idea of using program slicing [57] for extraction of CCCs was briefly discussed by Van Deursen et al. in [8]. In [15], Ettinger and Verbaere describe how program slicing, as used for method or object extraction, can be used to extract aspects. A program slice singles out all statements that may have affected the value of a given variable at a specific program point.

To extract a program slice to advice, object-oriented refactorings can be applied first, to make pointcut definition easier. Next, extraction of a slice is done in the same way as object extraction, only to a new advice instead of a class. No study on the general applicability of this approach to aspect-introducing refactoring is mentioned, other than a simple example.

Granularity

Although the technique of program slicing is only theoretical, the granularity would be comparable to that of catalogue-based refactorings, acting on method level. A problem again with these refactorings is that they do not map properly to the actual CCCs they are supposed to migrate to aspects.

3.1.4 Laws and rules

Cole and Borba describe laws to transform object-oriented code to aspects [6]. Laws are description of very small changes, that are guaranteed to preserve behaviour if given pre-conditions are met. Laws are bi-directional and are not intended to increase code quality on their own. They are supposed to be used to build larger refactorings based on the laws. Since laws are behaviour preserving, the refactorings composed of them also are. A list of example refactorings build from laws is also given in [6].

Similar to the laws introduced by Cole and Borba, Binkley et al. present rules in [2]. They propose a more concrete four-step iterative approach to refactor concerns using a set of rules. These rules are also behaviour preserving, but unlike laws, not bi-directional.

The rules are implemented using the TXL language [7]. TXL support the definition of grammar-based rules to perform source-to-source transformations. The rules have a
3.2 Tool Support for Refactoring

The method presented in this thesis for automated refactoring of sorts from object-oriented to aspect-oriented code is not the first attempt at tool support for aspect-introducing refactoring. There are a few other tools that help the refactoring, and we will discuss them here. This list contains all the tools we could find information on that will aid in the removal of CCCs from object-oriented code and introduction of aspects. Since this concerns state-of-the-art tools, most are only discussed in research papers as prototype systems to function as a proof-of-concept.

3.2.1 Role-based refactoring tool

Hannemann et al. describe a tool they used as proof-of-concept in [25] for role-based refactoring described in Section 3.1.1. The tool works for Java and is written as a plug-in for Eclipse. It uses the abstract syntax tree and Java models produced by the Java Development Tools plug-in. This role-based tool is not publicly available.

It contains an AutoMapper part that tries to match the CCC structure with the structure of the source code based on an initial mapping provided by the developer. This iteratively creates a complete mapping based on developer guidance. The refactorings are implemented as Eclipse refactorings (extending the abstract Refactoring class of Eclipse) in the Refactoring Engine part. The Impact Analysis part of the tool checks instructions of the Refactoring Engine part. If decisions need to be made the developer is asked to resolve them. The Impact Analysis part also ensures certain preconditions are constantly met.

A case study performed on JHotDraw shows successful refactoring of several CCCs known to be present, such as the Observer pattern.
3.2.2 Interface migration tool

Tonella and Ceccato present a tool in [55] as a proof-of-concept for the migration of interface implementations from classes to aspect in Java and AspectJ. Their tool, not publicly available, has a part that mines the code and a part that does the refactoring, both written in TXL [7]. The mining tool determines classes and interfaces to be refactored. The refactoring tool uses a module called UNPLUG to migrate the interface implementation to an aspect. The tool uses the declare parent construct of AspectJ as the main way of aspectising interface implementations.

Application of the tool to three packages of Java itself showed the tool and hence the concept works, but manual refinement of the resulting aspects was still necessary, as discussed in [55].

3.2.3 Rule-based refactoring tool: AOP-Migrator

Binkley et al. have created a tool for use with rule-based refactoring [2]. The tool automates all four steps described in Section 3.1.4 but allows for human guidance if necessary. The tool, called AOP-Migrator\(^1\) [3], is implemented as an Eclipse plug-in. The tool can be used to mark CCCs in the code, but suggests Dynamo\(^2\) for aspect mining. Once code is marked, the tool can extract it semi-automatically from the object-oriented code and migrate it to an aspect. The developer is required to select the target aspect and the refactoring (from a list of refactorings). Results based on four medium-sized case studies show aspects can be migrated automatically from existing code to aspects, although some code needs to be pre-processed by object-oriented refactorings. The tool is publicly available but has limitations because of the strict requirements on older versions of Eclipse (a development framework) and AJDT (an AspectJ toolkit for use in Eclipse).

3.3 Discussion of Limitations

The techniques and tools presented in the previous sections have limitations. The main limitations are discussed in this section and will be addressed by the technique and tool presented in the coming chapters.

3.3.1 Limitations of Methods

The discussed methods for aspect-introducing refactoring have several limitations. One is that they have inconsistent granularity, i.e. the techniques do not migrate exactly one CCC and the techniques differ in how much of a CCC is migrated with one refactoring. This inconsistency is partly due to the lack of use of a consistent definition of CCC.

A problem of the design pattern and role methods in Section 3.1.1, is that they are not suitable for concerns that fall in a different category. Contract enforcement, for example, is

\(^1\)http://se.itc.it/aop-migrator/
\(^2\)http://star.itc.it/dynamo/
a crosscutting concern that is neither a design pattern nor a role, and will therefore not be refactoring by those methods.

Another problem with the design pattern method from Section 3.1.1, is flexibility. Although design patterns are templated solutions to common problems [19], the implementation can be quite different from one instance to the next. This makes it difficult to show for design pattern implementations in general, what steps need to be taken in order to refactor the complete implementation to aspects.

Catalogue-based methods discussed in Section 3.1.2, have the disadvantage that they describe generic actions for migrating parts of a concern without showing how concrete concerns can be expressed in terms of these generic refactoring actions.

Program slicing from Section 3.1.3 is only theoretical, but a disadvantage would be that they only apply to specific concerns, namely those capturable by program slices. Role superimposition is a concern that would not fall in this category.

The behaviour preserving properties of laws and rules, Section 3.1.4, is a strength of those approaches. The problem is that this preservation only takes place when certain preconditions are met. Making sure these preconditions are indeed met can be difficult, especially when the fine-grained laws or rules are combined to form more concrete refactorings. In those combined refactorings, the set of preconditions to be met may grow large and may require several object-oriented refactoring to prepare the code before refactoring can take place.

### 3.3.2 Limitations of Tools

The main limitation of aspect-introducing refactoring tools is the availability of them. There is only one known tool publicly available, AOP-Migrator, which is dependent on old versions of Eclipse and AJDT. Other tools are only described in literature as proof-of-concept tools used to perform case studies.

Besides availability, there are other problems with the tools. One is that the results of aspect mining used as input for the refactoring are inconsistent. The tools either rely on one specific type of mining results, as is the case for interface migration tool, or they require the developer performing the refactoring to make the translation from mining results to refactoring input. Relying on one mining technique will most likely miss certain CCCs, making the tool incomplete. Requiring the developer to make the mapping is error prone and tedious, but inevitable if no consistent output is produced by the mining techniques.
Chapter 4

Sort-based Aspect-Introducing Refactoring

4.1 Introduction

In this chapter the proof-of-concept tool SAIR\(^1\) is introduced. The generic algorithm of the tool will be presented first, independently of the implementation. With ‘generic’ we mean the algorithm without the details that are specific to the sort being refactored. After describing the generic algorithm, an overview of the implementation will be given (still not specific to a sort) as well as a guide to how to use the tool in general.

SAIR is able to refactor CCCs, expressed in terms of sort instances, to aspects. The refactoring of two sorts is supported: the Role Superimposition (RSI) sort and the Consistent Behavior (CB) sort. The algorithm and details of the RSI sort refactoring are presented in Chapter 5. The algorithm and details of the CB sort refactoring are presented in Chapter 6.

These two sorts are chosen because they are two representative examples of sorts. They have different aspect solutions (inter-type declarations for the RSI sort and pointcuts and advice for the CB sort), they have a clear object-oriented idiom, they are relatively common and they are supported by SoQUET. The support by SoQUET is not relevant for the algorithm of SAIR, but it is relevant for the implementation.

The goal of SAIR is to demonstrate that aspect-introducing refactoring can be based on sorts and can be supported by a tool. To determine if the goal is reached, a case study is performed, the details and results of which are presented in Chapter 7.

4.2 Introductory Example

Before going into the details of the refactoring algorithm, let’s look at an example refactoring that SAIR will be able to perform. This example will provide a frame of reference when presenting the algorithm. The example is given in Java and the target aspect-oriented language is AspectJ. We will show how an RSI sort instance will appear in the Java code

\(^1\)Sort-based Aspect-Introducing Refactoring
4.2 Introductory Example

An RSI sort instance provides two explicit pieces of information. First, it provides a role interface, i.e. the interface that indicates the role that is imposed. Second, it provides a set of imposees that have the role imposed, i.e. the classes that implement that specific interface.

4.2.1 Situation before refactoring

Assume there is an RSI sort instance that documents the role of Storable to be imposed on a set of classes, indicating these classes can be written to and read from a file. These classes have other, primary functionality, so the imposing of the Storable role indicates a secondary concern.

The Storable interface is shown in Listing 4.1 and contains the two methods of the role, one for reading and one for writing. An example class implementing the interface is shown in Listing 4.2. This class reads and writes its class specific information—in this simple example only the radius. Other implementers of the Storable interface will perform similar actions.
4.2 Introductory Example

Since the role is a secondary and indeed crosscutting concern, it makes sense to implement it using an aspect. All that is needed, is a target aspect to implement the concern and an algorithm that describes how to migrate the implementation from the code to the aspect.

The algorithm for this example simply migrates the concrete implementations of the concern-related methods, i.e. the methods declared by \texttt{Storable}. This means moving the implementations from the original imposees to inter-type methods in the target aspect.

The indication that the \texttt{Storable} interface is implemented, is also migrated from the imposees to the aspect. This is done by declaring the interface a parent of all the imposees in the aspect and removing the interface from the list of implemented interface in each of the imposees. This way, the implementation of the storable concern is localised in one aspect and is no longer an implementation burden for the classes that are storable.

4.2.3 Situation after refactoring

After refactoring, the target aspect will look like Listing 4.3. This aspect now implements the concern of the \texttt{Storable} role. It declares the storable classes to implement the \texttt{Storable} interface and implements this interface for those classes using inter-type methods. The implementation of the implementers is much cleaner now, as shown for \texttt{Circle} in Listing 4.4.

**Listing 4.3: Aspect for \texttt{Storable} role after refactoring**

```java
package example.figures;
public aspect StorableFigureRole {
    declare parents: Circle implements Storable;
    // ... [other storable classes] ...
    // Storable Circle
    public void Circle.write(Output out){
        out.writeFloat(radius);
    }
    public void Circle.read(Input in){
        radius = in.readFloat();
    }
    // ... [other Storables] ...
}
```

**Listing 4.4: Example \texttt{Circle} after refactoring**

```java
package example.figures;
public class Circle {
    private float radius;
    // ... [primary concern] ...
}
```
4.2.4 Problems

Simple though this example may seem, there is a catch. The aspect as shown in Listing 4.3 will have compile errors. The reason is that the inter-type methods use fields that are private to the classes—radius in the case of Circle. Since the aspect is not privileged, these fields cannot be accessed from the aspect (see Section 2.2.2), resulting in compile errors.

For this example, a solution is simply to make the aspect privileged. Another solution can be to make the field public, or create and use public accessor methods for the field. So even such a simple problem has several possible solutions.

Less obvious and more complex problems can possibly occur and some may even block the refactoring. These problems may require more elaborate solutions. These problems should be resolved by the developer during the refactoring to let the refactoring to complete successfully.

The rest of this chapter will be going into the details of how a refactoring such as presented with this example can be performed. We look at the required input for the refactoring, how the problems can be resolved and how the actual refactoring can be performed.

4.3 Generic Algorithm

The algorithm of SAIR can be explained independently of the implementation. The generic version of that algorithm, i.e. the algorithm without the details for a specific sort, is described in this section. The sort-specific details of the refactoring algorithm for the two supported sorts, Role Superimposition and Consistent Behavior, will be presented in Section 5.2 and Section 6.2, respectively.

Although the generic algorithm steps (and the sort-specific steps) can be described independently of the implementation, these steps do assume an object-oriented setting and a target AOP language that is at least similar to AspectJ. The description of the algorithm uses classes, interfaces, methods and fields as they can be found in OOP languages like Java. Furthermore, AOP constructs like pointcuts, advice, inter-type declarations and privileged aspects are assumed to be available in the target AOP language.

The generic algorithm is presented as follows. We will first look at the input to the algorithm in Section 4.3.1. Next, we will see how the problems that may arise during refactoring are handled in Section 4.3.2. Finally, in Section 4.3.3 we will present a pseudocode version of the generic algorithm and the steps that are not specific to the sort.

4.3.1 Input to the algorithm

The refactoring algorithm requires input to determine which parts of the code need refactoring, how these parts need to be refactored and what the target of the refactoring is. The largest part of this input is provided by the sort being refactored, the rest needs to be provided by the developer.
Input provided by the sort

A sort instance describes the relation between the code elements that make up the CCC. These elements represent the context of the refactoring, i.e. the elements that need to be refactored. How these elements need to be refactored depends on the specific sort and the relation it imposes on the elements. The sort acts as a selector of the refactoring algorithm, i.e. the algorithm largely depends on the sort given as input.

In the introductory example of Section 4.2, the RSI sort instance was the input to the refactoring. This sort instance indicated the role interface and the imposees as context elements.

Input provided by the developer

The input provided by the sort is not sufficient to complete the refactoring. The refactoring algorithm needs at least an aspect as input that will serve as a target of the refactoring. After refactoring, the aspect will contain the CCC represented by the sort instance being refactored.

The target aspect can be an existing aspect that already has contents, or a newly created aspect. Existing aspects might already contain constructs like inter-type declarations, pointcuts and advice.

It may appear to break the one-concern-one-module principle advocated by AOP’s concern modularisation, to migrate sort instances to aspects that already contain code and therefore probably concerns. But although sorts are atomic, concrete CCCs can be built from more than one sort instance. For example, the Subject in the Observer pattern has two CCCs, described by an RSI sort and a CB sort (see Section 2.1 for an example Subject). The RSI sort imposes the Subject role on the subject and the CB sort represents the CCC of consistently notifying the Observers after a change to the Subject. These two sort instances could indeed be refactored to one aspect to represent the complete subject concern.

Besides the input of the target aspect, some additional input may be needed for the refactoring to be applied. This additional input is sort-specific and is discussed for the two supported sorts in the next two chapters.

4.3.2 Determining refactoring problems

In some cases, the information provided with the input is not enough to successfully apply the refactoring algorithm. Some problems can occur that might lead to compile errors after the refactoring or prevent part of the context from being refactored. In the example of Section 4.2, the use of private fields from the unprivileged aspects leads to compile errors.

Such problems can only be determined after the input has been given (since then the context is known), but do require the intervention of the developer as soon as they are discovered, to prevent unexpected problems during or after refactoring.

The algorithm of SAIR solves this issue by detecting potential problems before they occur. As soon as all input information is gathered, the problems that will occur are deter-
mined, along with a set of solutions that will resolve the problems. This allows the developer performing the refactoring to solve these problems before the refactoring continues.

**Problems**

Refactoring problem only occur when certain problem-conditions are met. How these conditions relate to concrete problems depends on the sort being refactored. Not all refactoring problems can occur for all sorts. The refactoring problems for the two supported sorts and how they can occur, will be presented in the next two chapters.

In the introductory example, the visibility problems occurred because of two conditions: (1) private fields were referenced from the aspect and (2) the aspect was not privileged. All possible problems have such conditions that will, when met, make the problem occur.

The following is a list of all refactoring problems currently detected and supported by SAIR, and the conditions under which they occur.

**Access of private method**  If the aspect will access a private method after the refactoring, but the aspect is not privileged, this problem is detected.

**Access of private field**  If the aspect will access a private field after the refactoring, but the aspect is not privileged, this problem is detected.

**Access of package-visible method**  If the aspect will access a package-visible method after the refactoring, but the aspect is not privileged and not in the same package as the method, this problem is detected.

**Access of package-visible field**  If the aspect will access a package-visible field after the refactoring, but the aspect is not privileged and not in the same package as the field, this problem is detected.

**Access of protected method**  If the aspect will access a protected method after the refactoring, but the aspect is not privileged and not in the same package as the method, this problem is detected.

**Access of protected field**  If the aspect will access a protected field after the refactoring, but the aspect is not privileged and not in the same package as the field, this problem is detected.

**Access of a private inner class**  If the aspect will access a private inner class, but the aspect is not privileged, this problem is detected.

**Access of a package-visible inner class**  If the aspect will access a package-visible inner class, but the aspect is not privileged and not in the same package, this problem is detected.

**Access of a protected inner class**  If the aspect will access a protected inner class, but the aspect is not privileged and not in the same package, this problem is detected.
Access of a package-visible class  If the aspect will access a package-visible class, but is not in the same package, this problem is detected.

Access of super field  If the aspect will access a super field explicitly from an advice body, this problem is detected.

Access of super method  If the aspect will access a super method explicitly from an advice body, this problem is detected.

Advising nested call  If a advice needs to be created for a statement that is nested in the method body (so not directly in method body but e.g. in an if block), this problem is detected.

Advising neither first nor last call  If a advice needs to be created for a statement that is neither the first nor the last statement in a method body, this problem is detected.

Access of local variable  If the aspect will need to access a variable local to the method (but not a method parameter) in an advice body, this problem is detected.

Solutions

To resolve the problems that are detected after the input is provided, a set of solutions is determined for each problem. The detected problems are then resolved by letting the developer select one of the several possible solutions per problem. Resolving the problem is not exactly the same as solving the problem. By resolving we mean selecting a solution for the problem from the list of determined solutions. Not all solutions actually solve the problem, since some solutions will simply make the algorithm ignore or skip part of the context.

Once all problems have been resolved (i.e. all problems have a selected solution) the actual refactoring can commence. The refactoring algorithm applies these solutions before the actual refactoring starts, to make sure the refactoring can finish without unexpected compile errors or issues.

The solutions for the problems have several properties:

Solution Type  This is a unique type indicator for each solution. The list of solutions that is determined for each problem, can contain at most one solution of each solution type.

Requires refresh  This indicates whether the set of problems should be updated (TRUE) or not (FALSE) when this solution is selected for a problem. This will be TRUE for solution that possibly introduce new problems or eliminate other problems when they are applied.

Exclusive  This indicates whether selecting this type of solution for one problem makes it unavailable for other problems (TRUE) or not (FALSE). This will be TRUE for solutions that can only be applied for one problem.
Propagate  This indicates whether selection of a solution will propagate to other problems (TRUE) or not (FALSE). Propagation is based on equivalence of solutions: solutions are equivalent if they are of the same solution type and work on the same data. When a solution that propagates is selected, the equivalent solution is selected for all determined problems that have no selected solutions yet.

The following is a list of all the solutions for refactoring problems and their values for the solution properties.

Ignore  This solution will ignore the problem. Applying this solution will probably lead to compile errors, but the developer may choose to take care of them after the refactoring. If the problem cannot be ignored without blocking the refactoring, the direct culprit of this problem is excluded from the refactoring when this solution is applied.

Make aspect privileged  This solution will make the aspect privileged. Applying this solution will alleviate any visibility problems with methods, fields and inner classes. It propagates, so selecting this solution for one problem, will automatically select the solution of this type for all problems that have no selected solution yet but do have an equivalent solution.

Make member public  This solution will make a given member public. Applying this solution will alleviate any visibility problems with the given member (method, field or inner class), since the aspect can always access a public member. It propagates, so selecting this solution will automatically select equivalent solutions for unresolved problems.

Make class public  This solution will make a given class public. Applying this solution will alleviate any visibility problems with the given top-level class, since the aspect can always access a public class. It propagates, so selecting this solution will automatically select equivalent solutions for unresolved problems.

Move aspect to package  This solution will move the aspect to a different package. Applying this solution will alleviate any package-visible and protected visibility problems for classes and members in the target package, since the aspect is now in the same package. It requires a refresh of the problem list, because moving the aspect might
lead to new visibility problems (with the originating package), or might solve other visibility problems for the target package. This solution is exclusive: if it is selected for one problem, it cannot be selected for another (the aspect can only be moved to one other package).

**Refresh:** Exclusive: Propagate: Solution type:

<table>
<thead>
<tr>
<th>Refresh</th>
<th>Exclusive</th>
<th>Propagate</th>
<th>Solution type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>MOVE_ASPECT_TO_PACKAGE</td>
</tr>
</tbody>
</table>

**Create and use getter/setter** This solution will create a public getter and/or setter for a field and replace access to that field with the getter/setter. Applying this solution will alleviate any visibility problems, since the aspect now uses public accessors. It propagates, so selecting this solution will automatically select equivalent solutions for unresolved problems.

**Refresh:** Exclusive: Propagate: Solution type:

<table>
<thead>
<tr>
<th>Refresh</th>
<th>Exclusive</th>
<th>Propagate</th>
<th>Solution type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
<td>CREATE_GETTER_SETTER_FOR_FIELD</td>
</tr>
</tbody>
</table>

**Exclude method from refactoring** This solution will exclude a given method from the refactoring. This method will be the culprit of the problem, and excluding it from the refactoring resolves the problem. Applying this solution requires a refresh of the problems, since excluding this method may eliminate other problems that have the same method as source of the problem.

**Refresh:** Exclusive: Propagate: Solution type:

<table>
<thead>
<tr>
<th>Refresh</th>
<th>Exclusive</th>
<th>Propagate</th>
<th>Solution type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>EXCLUDE_METHOD</td>
</tr>
</tbody>
</table>

**Exclude class from refactoring** This solution will exclude a given class from the refactoring. This class will be the culprit of the problem, and excluding it from the refactoring resolves the problem. Applying this solution requires a refresh of the problems, since excluding this class may eliminate other problems that have the same class as source of the problem.

**Refresh:** Exclusive: Propagate: Solution type:

<table>
<thead>
<tr>
<th>Refresh</th>
<th>Exclusive</th>
<th>Propagate</th>
<th>Solution type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>EXCLUDE_CLASS</td>
</tr>
</tbody>
</table>

**Advise as after** This solution will create after advice for a given statement, even if that statement did not originally come last in or first after the captured join point.

**Refresh:** Exclusive: Propagate: Solution type:

<table>
<thead>
<tr>
<th>Refresh</th>
<th>Exclusive</th>
<th>Propagate</th>
<th>Solution type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>ADVISE_AS_AFTER</td>
</tr>
</tbody>
</table>

**Advise as before** This solution will create before advice for a given statement, even if that statement did not originally come last before or first in the captured join point.

**Refresh:** Exclusive: Propagate: Solution type:

<table>
<thead>
<tr>
<th>Refresh</th>
<th>Exclusive</th>
<th>Propagate</th>
<th>Solution type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>ADVISE_AS_BEFORE</td>
</tr>
</tbody>
</table>
Copy super body to advice  This solution will copy the body of a super method to advice, replacing a direct call to that super method. Applying this solution requires a refresh of the problems, since using the body of the super method may create new problems.

Refresh:  Exclusive:  Propagate:  Solution type:
TRUE   FALSE   FALSE   COPY_SUPER_BODY_TO_ADVICE

Create super redirect method in aspect  This solution creates a method in the aspect that acts as a redirect method for calling a super method. This will replace direct calls to super methods in an advice body. Since inter-type declarations can access super methods, this alleviates problems with calling super methods from advice.

Refresh:  Exclusive:  Propagate:  Solution type:
FALSE  FALSE   FALSE   SUPER_REDIRECT_METHOD_IN_ASPECT

Make local variable a field  This solution replaces a local variable in a method with a private field in the surrounding class. This alleviates problems with access to local variables in advice. Applying this solution requires a refresh of the problems, since the new use of a private field may create visibility problems.

Refresh:  Exclusive:  Propagate:  Solution type:
TRUE   FALSE   FALSE   MAKE_LOCAL_FIELD

Solutions per problem
The solutions per problem and the actions of applying the solutions can depend on the specific sort being refactored. For a complete list of solutions per problem and the action in the context of a specific sort, see Appendix B. There are, however, two types of solutions applicable to all problems.

The first type of solution applicable to all problems is the IGNORE solution. Selecting this solution will simply ignore the problem and will most likely result in compile errors. If the refactoring cannot be performed when the problem is simply ignored, the IGNORE solution will exclude part of the context—the part containing the direct culprit of this problem—from the refactoring.

The second type of solution applicable to all problems, is a solution that will explicitly exclude part of the context responsible for the problem. This is done with either an EXCLUDE_METHOD solution or an EXCLUDE_CLASS solution. This is convenient for problems that require some more attention, problems that are caused by context elements mistakenly captured by the sort instance being refactored or problems that are caused by improper implementation.

4.3.3  Refactoring steps
The generic version of the refactoring algorithm is presented in pseudo-code in Algorithm 4.1. The initial input required by the algorithm is the sort instance being refactored. When this algorithm is applied, the following steps are taken.
Algorithm 4.1 Main refactoring steps

1: procedure MAIN_REFACTORING_ALGORITHM(sort)
2:  input ← GATHER_INPUT(sort)
3:  problems ← DETERMINE_PROBLEMS(input)
4:  input.problems ← problems
5:  if problems.size > 0 then
6:     RESOLVE_PROBLEMS(input)
7:     for all problems as problem do
8:       APPLY SELECTED SOLUTION(problem)
9:     end for
10:  end if
11:  PERFORM_MIGRATION(input)
12: end procedure

Line 2: gather input Here the developer input is collected, which is partly dependent on the specific sort being refactored. As described in Section 4.3.1, input to the refactoring is a combination of the sort instance (an external parameter of the algorithm) and the developer input.

Line 3: determine problems As described in Section 4.3.2, this step determines the problems that require resolving by the developer before refactoring can commence. The details of this step are dependent on both the specific sort that is refactored as well as the implementation.

Line 6: resolve problems This step lets developer resolve the problems by selecting one of the possible solutions. If no problems were determined, this step is skipped.

Lines 7 - 9: apply solutions In this step, the solutions selected in the previous step are applied. How these are applied depends on the solution and possibly on the specific refactoring.

Line 11: perform migration This is the largest step: based on the input provided by the sort instance and the developer, the concern represented by the sort is migrated from the original context to the target aspect. The actual algorithm for this step depends on the specific sort.

The algorithms of two of these steps, GATHER_INPUT (line 2) and PERFORM_MIGRATION (line 11), will be elaborated for the RSI and CB sorts in Section 5.2.3 and Section 6.2.3, respectively.

Not all procedures in the algorithms presented in this chapter and the following two are completely detailed. In those cases, the procedures are briefly explained in the text, or have obvious meanings. The concrete algorithm of such a procedure is not relevant to the migration step of the algorithm (e.g. RESOLVE_PROBLEMS) or is very implementation dependent (e.g. DETERMINE_PROBLEMS).
4.4 Implementation

This section presents the main outline of the implementation of SAIR. We will look at the dependency of SAIR on SOQUET, explain how the Eclipse framework is used and introduce the main architecture of the tool. We discuss the main problems and difficulties that were encountered and how they were dealt with.

4.4.1 Dependency on SOQUET

As mentioned in Section 2.4, SOQUET is a tool that allows developers to build a concern model using sort instances. This concern model can be used as a starting point for refactoring. The sort instances in the model provide exactly the right input of context elements and their relations, as described in Section 4.3.1.

Therefore, SAIR uses SOQUET to provide the sort instance input for refactoring. The current version of SOQUET is only for use with programs written in Java. Consequently, SAIR is also for use with programs written in Java. With Java programs to refactor, the obvious choice for a target aspect language is AspectJ [1] (see Section 2.2.2).

4.4.2 Use of Eclipse

Eclipse [10] is an open-source software framework written primarily in Java. The capabilities of Eclipse can be extended by installing plug-ins written for Eclipse, such as development toolkits for various programming languages. The most popular distribution of Eclipse is that of software development kit (SDK) for Java, where the Java Development Tools plug-in provides an integrated development environment (IDE) for Java. This is also the distribution of Eclipse used by SAIR.

Eclipse can be extended by writing custom plug-ins to provide new functionality. SOQUET is such a plug-in—it extends the Eclipse framework by providing the functionality discussed in Section 2.4.

SAIR is also implemented as an Eclipse plug-in and adds the capability of aspect-introducing refactoring. Eclipse was chosen for the following main reasons:

1. SAIR uses SOQUET, which is implemented as an Eclipse plug-in;
2. The Eclipse SDK provides an extensive set of tools to query and manipulate Java source code and models built from the source code (Java Development Tools);
3. The Eclipse SDK provides the Language Toolkit, a toolkit that supports the creation of refactorings that look and feel like native Eclipse refactorings;
4. The Eclipse SDK can have support for AspectJ, in the form of the AspectJ Development Tools plug-in.

Figure 4.1 shows the architecture of the Eclipse SDK. Besides the Java Development Tools, Plug-in Development Environment, Language Toolkit and AspectJ Development Tools it also shows how SOQUET and SAIR are connected to the platform and each other.
Plug-in Development Environment

The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and rich client platform (RCP) products [12]. SAIR is implemented as a plug-in for Eclipse using the PDE.

Plug-ins can provide extension points where other plug-ins can contribute functionality. Using such an extension point, SAIR contributes a ‘refactoring’ action to the context menu of the Concern Model view, defined by SOQUET. This action appears in the context menu of single sort instances. Any action that is chosen from that menu, including the refactoring action contributed by SAIR, automatically gets the selected item as context information. This is a one-way dependency—SOQUET isn’t aware of the existence of SAIR or the addition of a context menu action.

Java Development Tools

The Java Development Tools (JDT) plug-in is the plug-in that implements the Java IDE which supports the development of any Java application. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. Since Eclipse and the plug-ins are written in Java, the JDT project allows Eclipse to be a development environment for itself [11].

SAIR is not only implemented using the JDT (and PDE), it also uses it internally. The JDT provides a set of features for parsing Java source code and building models from it, which can be used to query code and relations between code elements. It also provides tools to modify existing code without rewriting the source code directly.
Java Model  
Java models created by the JDT are tree structures, with an IJavaProject as root for each project. This Java project has IPackageFragmentRoots as children, which in turn have IPackageFragmentRoots as children. These package fragments can contain ICompilationUnits, which represent Java source files. The tree has ITypes (Java classes, interfaces and enums) inside compilation units, and IMethods, IFields and ITypes inside an IType. All these model IJavaElements maintain a relation with their parents and children if they have any. This makes it possible to navigate up and down the tree and thus through the project.

The model elements are shown in Figure 4.2. This figure shows clearly the tree-like structure of the Java model. It also shows that ITypes can occur inside other ITypes, representing inner classes.

ASTs and rewriting  
Another model provided by the JDT is the abstract syntax tree (AST). The AST is an abstract tree representation of a Java source file. It provides more detail than the Java model and is easily created by supplying an ICompilationUnit to a parser. Where the lowest level of elements in the Java model are IFields and IMethods, the AST contains information down to single expressions, identifiers and operands. ASTs are used in SAIR for two main purposes.

First, if bindings are requested during creation of the AST from a source file, all nodes in the AST that can refer to elements in the Java model, have a binding that links the node to that element. For instance, method call nodes have bindings to the IMethod that is called. This allows SAIR to find relations between ASTs from different source files using the project wide Java model.

Second, the JDT provides tools that record changes made to the AST and that are able to rewrite the respective source files to reflect those changes. This is done by reporting modifications to the AST to an ASTRewrite object, which will create a TextEdit when asked to rewrite the AST. This TextEdit can be applied to the file that was the source of the original AST to make it reflect the recorded changes. The rewriter also takes care of formatting the code. The rewriting mechanism provides an easy and consistent way to modify Java source code using the JDT.
**Working copies**  The Java Model provided by JDT supports a feature that is used by SAIR, namely the concept of *working copies*. A working copy is an in-memory representation of an `ICompilationUnit`. Changes made to the working copy are not reflected by the file on disc until the working copy is explicitly reconciled with the `ICompilationUnit` it was originally created for [13].

Working copies are used in Eclipse in general to ensure the most recent version is used when modifying code programmatically, e.g. when a file is open and modified in an editor but not yet saved. This means working copies can be *shared* between clients so that they all work on the same in-memory copy. Just like normal compilation units, working copies can also be parsed into an AST. SAIR uses working copies in this way.

SAIR also uses working copies in another way, in the problem resolving step. When applying solutions, some of these solutions modify code. Instead of modifying the original code directly, these solutions modify working copies. The migration step then uses those modified working copies instead of the on-disc versions. The modifications of the migration step are added to the changes from the problem resolving step. Finally all accumulated changes are applied to the original `ICompilationUnit` when finishing the refactoring.

**Language Toolkit**

One of the tools supplied by Eclipse is the Language Toolkit (LTK). It is the start of a generic, language independent tooling layer for Eclipse. It currently allows developers to create refactorings that look and feel just like other refactorings in Eclipse, or that participate in existing refactorings.

SAIR uses the LTK to provide the aspect-introducing refactoring. This is a combination of the refactoring wizard to interact with the developer, and a mechanism to perform the refactoring.

One feature of refactorings written using the LTK, is that the wizards automatically provide a preview of the changes to be made before they are actually applied. This allows the developer to review what is about to happen and possibly go back in the wizard and make some modifications to the input.

A simplified class diagram showing the relevant class structure of the LTK is shown in Figure 4.3. This diagram shows the basic classes provided by LTK and used by the implementation of SAIR. The use of LTK can be split up in two parts: the user interface part and the refactoring part.

Figure 4.4 shows how SAIR uses the LTK to implement the refactoring wizard. The `RefactoringWizard` is extended using an `AspectIntroductionWizard`. This abstract wizard is implemented further for each sort. The `UserInputWizardPage` is extended by a problem resolving page (`ResolveProblemsPage`) and an abstract input page (`AbstractInputPage`) that is implemented further for each sort to allow for sort-specific input.

Figure 4.5 shows how SAIR uses the LTK to implement the refactoring itself. The abstract `Refactoring` class provided by LTK is extended and implemented by SAIR.

---

2The LTK version used for SAIR is 3.2 (included in Eclipse 3.2.2)
4.4 Implementation

Sort-based Aspect-Introducing Refactoring

Figure 4.3: Simplified class diagram of relevant classes from LTK

Figure 4.4: Simplified class diagram showing relation between LTK and SAIR for wizards

with the AspectIntroducingRefactoring. This refactoring delegates to concrete instances of the abstract RefactoringDelegate. For each sort instance, there is one concrete delegate class that implements the required methods to perform the refactoring.

Figure 4.5 also shows how Changes are used by SAIR. The abstract Refactoring is expected to create a Change and the concrete AspectIntroducingRefactoring fulfils this expectation by creating a CompositeChange. This change is made up of different changes created by the RefactoringDelegate, most of which are CompilationUnitChanges. These CompilationUnitChanges are subclasses of the TextEditBasedChange and can therefore be given the TextEdits produced by the JDT AST rewriter.
Figure 4.5: Simplified class diagram showing relation between LTK and SAIR for refactoring

AspectJ Development Tools

The AspectJ Development Tools are the AspectJ complement of the JDT [1]. AJDT makes Eclipse an AspectJ IDE which facilitates the development of aspect-oriented programs using AspectJ.

AJDT is as flexible and easy to use as JDT for programming in AspectJ. However, the limitations are encountered when writing a plug-in that uses AJDT to inspect and modify aspects programmatically. The main problem is that the AJDT is not yet finished\(^3\), providing stub implementations for part of the functionality.

As a consequence, the AJDT cannot rewrite all aspect-specific elements of the AjAST created by the AspectJ parser. This means that the aspect code generation for inter-type declarations, pointcuts and advice is not done using a rewriter that tracks changes and creates TextEdits that will reflect those changes. It is done by manually building those TextEdits when the AjASTRewrite class is not able to write them.

There are some other small consequences with the incompleteness of the AJDT. For instance, previews in the wizard cannot be highlighted correctly for AspectJ code, making some AspectJ keywords look like normal identifiers.

Although the AJDT is not finished and cannot be used to rewrite the aspect structures, it is still used by SAIR. It is used to parse the target aspect, something the JDT parser cannot do. The AjAST can still be used to modify non-aspect elements, in particular the package.

\(^3\)The version of AJDT used for SAIR is 1.4.2, which includes AspectJ 1.5.4
4.4 Implementation Sort-based Aspect-Introducing Refactoring

4.4.3 Architecture overview

The architecture of SAIR consists of four subsystems, shown in Figure 4.6.

**User interface subsystem**  The user interface subsystem is used to provide a wizard to the developer to allow for input. It is also used to show the list of problems and allows the developer to resolve them. Finally, this subsystem is used to provide a preview of all changes that will be made before they are actually applied.

**Code modification subsystem**  This subsystem is responsible for creating modifications to the actual Java and AspectJ code. Since both the problem resolving subsystem and the refactoring subsystem modify code, they both report modifications to this subsystem. The subsystem creates changes that represent the code modifications. These changes are fed back into the refactoring subsystem.

**Refactoring subsystem**  This subsystem is the controller of the refactoring. It handles the input, controls the determining of problems and application of solutions and finally performs the refactoring. The result of the refactoring is a Change object that reflects all changes to the workspace that need to be made for the refactoring to be performed. This Change is passed back to the input subsystem for preview. Finally, the LTK takes care of applying the Change if the developer agrees and finishes the refactoring.

**Problem subsystem**  This subsystem is responsible for both determining problems as well as the application of solutions. It determines problems based on the input passed...
by the refactoring subsystem. When the problems have been resolved by the developer, this subsystem applies the solutions and reports the modifications to the code modification subsystem.

Information passing

The input provided by the sort and the developer needs to be passed between the wizard, the refactoring subsystem and the problem subsystem. The problem subsystem itself will possibly modify the input information when applying solutions. The IRefactoringInfo interface provides an interface for information classes used to pass information between subsystems. The AbstractRefactoringInfo class provides a base implementation for the generic information and is extended by classes specific to the sort being refactored.

Figure 4.7 shows the abstract information class with the most important information it contains. This includes the target aspect and the Java project that is being refactored. The information class also keeps track of excluded methods and classes. This is used by the problem subsystem when applying a solution that excludes part of the context. Finally, the abstract information class has a ProblemManager that keeps track of all the problems in the current refactoring.

4.4.4 Design issues

Anonymous classes

Anonymous classes require special treatment when encountered in a refactoring. The problem is that the lack of a name for these classes makes it impossible to reference them from aspects. Creating pointcuts, advice or inter-type declarations for such classes is not possible. The current version of SAIR therefore automatically excludes all anonymous classes that need to be referenced from the aspect after refactoring.

Internal code

In Eclipse, plug-ins can by default access all public classes of other plug-ins, as long as the packages are exported. Developers can choose to restrict access to their plug-in by not
including packages in the exported packages list. These ‘forbidden’ packages are not visible to other plug-ins and generate compile errors when used.

Another method of access restriction is declaring packages as hidden. These packages are typically ‘internal’ packages, i.e. not part of the public API of the plug-in. Such packages are visible from other plug-ins and can be used, but generate a warning. This warning states that the used class is not part of a public API but is internal and may change without notice.

Although discouraged, SAIR uses internal classes of the AJDT on two occasions, both in the wizard. The first use is a selection dialogue similar to that used when selecting superclass or interfaces in the ‘new class’ wizard is used. This selection dialogue is used to select the target aspect.

Another use of internal code concerns the ‘New aspect...’ button. The wizard that is opened when this button is pressed, is the default ‘New aspect’ wizard from AJDT, which can be opened without the use of internal code, simply by using the wizard ID. However, when the user finishes the aspect creation, it would be very nice if the fully qualified name of the created aspect is automatically entered in the target aspect field. This requires access to an internal subclass of the wizard.

Note that the first use can be circumvented by creating a custom selection dialogue or implementing content assist on the target aspect field, and that the second use of internal code can be circumvented by listening to resource changes and check if any new aspects are created. Both of these take time to implement and did not have the highest priority.

Limitations of the SOQUET API

When selecting a sort instance in the concern model view of SOQUET, the concrete class that is associated with that instance is a SortInstance. In fact, only for SortInstances will the SAIR refactoring option appear in the context menu.

These SortInstances have an ISortSearchQuery that represents the query on the source code indicating the context. Ideally, we would run this query and collect the results according to the sort. However, SOQUET currently does not provide an API to run these queries and collect the results in a straightforward and transparent manner.

The workaround used by SAIR relies on the ISortSearchQuerySpecification, which is present in each ISortSearchQuery. This specification contains the information that was entered when the sort instance was specified and depends on the specific sort. SAIR uses the RSISearchQuerySpecification when the instance is an RSI sort and the CBSearchQuerySpecification when it is a CB sort.

Now, when specifying the sort instance, instead of indicating the context in a broad manner, e.g. the project or the hierarchy of Command, we have to indicate the context elements as they would be found by SOQUET. These manually entered context elements can be found in the ISortSearchQuerySpecification without executing the query and therefore all necessary refactoring information can be collected.

As an example, consider the virtual ObservableCommand role (see Section 5.1 for more on virtual roles). This role indicates that all classes in the hierarchy of Command implement two methods from the Command interface related to observability: the method
addCommandListner() and the method removeCommandListner(). Normally we would indicate as a context the 'hierarchy of Command', but in the workaround we will enumerate all the implementers in the hierarchy of Command explicitly. So the context is AbstractCommand, UndoableCommand, etc.

4.5 Usage

Now that the algorithm and implementation of SAIR have been presented, it is time to see how SAIR works in practice. This section will show the generic steps of the refactoring by providing screenshots and explaining the various possible actions.

Only the input page of the refactoring wizards is different for each the two supported sorts. Other pages have different content for each sort and sort instance, but the actions and type of information are the same. The input pages for refactoring of RSI sorts and CB sorts are presented in Section 5.4 and Section 6.4, respectively.

This guide for using SAIR assumes the following situation. There is a Java project open in Eclipse with no compile errors. A concern model has been built for this project using SOQUET and this concern model is loaded in the ‘Concern Model’ view of SOQUET in Eclipse.

4.5.1 Starting refactoring in SOQUET

Starting the refactoring of a sort instance in SOQUET is straightforward. Simply browse to the sort instance of interest and right click on it to bring up the context menu shown in Figure 4.8.

This figure shows that the last option in the context menu is ‘Refactor CCC to aspect’. Clicking on this starts the refactoring by opening the refactoring wizard.
4.5 Usage

4.5.2 Providing input

The first page shown when the refactoring wizard is opened is the input page, shown in Figure 4.9. In this case it is the input page for a Role Superimposition sort refactoring, but the sort-specific input fields have been left out of the figure for sake of clarity.

There are a couple of things to notice about the input page in Figure 4.9. First of all, the only enabled navigation button in the bottom row of buttons, is ‘Cancel’, which will cancel the refactoring when pressed. Second, there is an input field for a target aspect, with a ‘Browse...’ button and a ‘New Aspect...’ button. Finally, there is a disabled and checked checkbox that excludes anonymous classes from the refactoring (see Section 4.4.4).

To enable the ‘Next’ button, at least a target aspect is needed as input and possibly sort specific input. There are three ways to provide a target aspect. The first is simply typing in the qualified name of the aspect. The second is pressing the ‘Browse...’ button, which brings up a dialogue to select an existing aspect. The third way is pressing the ‘New Aspect...’ button, which starts the AJDT wizard for creating new aspects. The newly created aspect is automatically filled in as the target aspect when the AJDT wizard finishes.

If a mistake is made when providing input to the refactoring, the wizard shows an error message, as shown in Figure 4.10. This message will disappear when the mistake has been rectified. In the case of Figure 4.10, the error message will disappear when an existing target aspect has been provided.

Figure 4.9: Providing the target aspect for the refactoring

Figure 4.10: Example of feedback on invalid input
Once all necessary input has been provided and no errors occur, the refactoring can continue. This is indicated by an enabled ‘Next’ button, as shown in Figure 4.11. Pressing the ‘Next’ button will determine the refactoring problems that will occur with the given input.

### 4.5.3 Resolving problems

When the refactoring problems that will occur have been determined, the wizard will present the problem page, shown in Figure 4.12. This page has three main parts of interest. The first is the list of problems that need to be resolved. Beneath that is the list of solutions that are available for the selected problem. At the bottom is a small screen that provides a little more information on the selected problem.

Besides the three main parts, there is a checkbox at the top of the page that says ‘Ignore all problems...’. Checking this checkbox is a shortcut to selecting the IGNORE solution for all problems. This can be used to quickly be able to browse to the preview to better assess the impact of ignoring problems.

In the list of problems, unresolved problems are shown with a red background. Once a problem has been resolved it is shown on a white background. Above the list is a small header that shows how many problems are still unresolved.

When a problem is selected, the set of available solutions for that problem is shown in the list beneath the problem list. Selecting a solution in this list will resolve the selected problem. If a solution is selected that propagates (see Section 4.3.2), possibly other problems will also be resolved automatically, if they have no selected solution yet but they do have a solution equivalent to the propagating solution. If a solution is selected that is exclusive, the same solution is no longer available for other problems. The unavailability of an exclusive solution that has already been selected for another problem is shown in Figure 4.13.

If a solution is selected that requires the set of problems to be refreshed, a dialogue is presented, shown in Figure 4.14. If ‘Cancel’ is pressed, the solution is not selected.
4.5 Usage

Sort-based Aspect-Introducing Refactoring

Figure 4.12: Page for resolving problems

Figure 4.13: Disabled exclusive solution, already selected for another problem
and nothing changes. If ‘OK’ is pressed, the set of problems is redetermined under the assumption that the solution is applied. Already resolved problems will still have a selected solution if they still appear in the refreshed set of problems.

Once all problems have been resolved, the ‘Next’ button and ‘Finish’ button are enabled as shown in Figure 4.15. Pressing the ‘Finish’ button will immediately apply the selected solutions and perform the refactoring. If the ‘Next’ button is pressed, a preview of the refactoring results is created.

### 4.5.4 Previewing changes

Previewing changes before they are applied is a default feature for refactoring wizards in Eclipse. The preview page for the sort refactoring, shown in Figure 4.16 should therefore be familiar to Eclipse users. It consists of two areas: a list of all changes to be applied and preview of the selected change if available.

The preview is only available for changes that modify source code. Changes that modify the Eclipse workspace in other ways, such as moving files, cannot be previewed. The preview consists of a before and after section. The before section on the left shows the current content of the source file that is modified. The after section on the right shows the
4.6 Summary

This chapter introduced the generic algorithm behind SAIR, a tool that is capable of refactoring sorts to aspects. The generic algorithm takes a sort instance as input.

The main steps of the algorithm are: (1) gathering input, (2) determining problems that will occur and the solutions that can be used to resolve them, (3) resolving the problems, (4) applying the selected solutions and (5) performing the migration of the sort instance to an aspect.

In Section 4.4, some general implementation details were discussed, such as the dependency on Eclipse and SOQUET. Finally, a short overview of using SAIR has been given in Section 4.5, accompanied by screenshots.

The following two chapters will show the details of the generic algorithm for the two supported sorts. These sort-specific details concern mainly the input step of the algorithm and the migration step.
Chapter 5

Refactoring Role Superimposition Sort

One of the two sorts implemented by SAIr is the Role Superimposition (RSI) sort. In this chapter we will see what the intention of refactoring an RSI sort instance is. This intention describes broadly how the refactoring will modify the code. The steps and details of the algorithm specific to the RSI sort are also presented, as well as an implementation overview of these steps. Finally, we show how input is provided when refactoring an RSI sort instance using SAIr.

The RSI sort is one of twelve sorts identified by Marin in [36, 32]. The RSI sort, as explained in Section 2.4.1, describes crosscutting concerns that indicate the implementation of a secondary role or responsibility. This secondary role or responsibility is tangled with the primary role of classes. Concrete instances of an RSI sort occur in object-oriented languages (such as Java) as sets of classes that implement a common role, usually an interface, besides the primary role of those classes. An example of an RSI sort instance is the Storable concern, discussed in Section 4.2.

5.1 Refactoring Intention of RSI Sort

To understand the RSI-specific details and steps of the algorithm, the intention of refactoring the RSI sort should be clear. This intention indicates broadly what code transformation are needed to create an aspect-oriented implementation for RSI sort instances. The intention will describe the input that is required and the resulting output. Such a broad overview will make it easier to understand the problems that need to be determined, and the algorithm steps.

The RSI sort defines two sets of elements. One is the set of classes that have a role superimposed on them, these are the imposees. The other set is the set of members that make up that role. These members are part of an interface or class that is the superimposed role.

When the set of members that make up the superimposed role are not declared in a dedicated interface or class (i.e. some members of the role interface or class do not belong
5.1 Refactoring Intention of RSI Sort

5.1.1 Simplifying assumptions about roles

Although in theory a superimposed role can be a class as well as interfaces, the algorithms in presented in this chapter, and therefore the implementation of SAIR are restricted to interfaces as roles only. This means only interfaces can be a role or contain the role members of a virtual role.

Additionally, only methods are considered as role members in the algorithms and implementation of SAIR. This further simplifies the roles to only interfaces with methods. Virtual roles consist of a subset of methods of an interface.

The rationale behind this small simplification of the algorithm and therefore the tool, is discussed in Section 5.3.2.

5.1.2 Refactoring RSI to aspect

The desired aspect language mechanism of the RSI sort is introduction [36]. This means introducing parents and members to classes by means of inter-type declarations. Using inter-type parent declarations, the role interface can be imposed on the imposees and using inter-type method declarations, the implementations of the role methods can be added to the imposees.

When refactoring an existing RSI sort instance to an aspect, these inter-type declarations need to be created for the imposees and role methods. In case the role is non-virtual, this means that the imposees no longer implement the role directly, but that role interface will be declared a parent of the imposees, inside the aspect. Also, the methods of the role are not implemented in the imposees, but are moved to the aspect as inter-type method declarations. This leads to an aspect-oriented implementation of the RSI sort instance.

5.1.3 Virtual roles

If the role is virtual, an option is to abstract the virtual role methods to a concrete new interface. The newly abstracted interface then becomes the role, but instead of being declared parent of all imposees, it will be declared parent of the interface that originally contained the virtual role. This will ensure original calls to role methods throughout the program are not broken.

5.1.4 Example refactoring

To further clarify the intention, the following example shows the before and after situation of refactoring an RSI sort instance with a virtual role. For a non-virtual role, an example refactoring was given in the previous chapter, in Section 4.2.

Suppose there is an interface called Command that contains a virtual role of being undoable. Listing 5.1 shows this interface with the virtual undoable-role indicated. An example imposee of this virtual role is shown in Listing 5.2.
The RSI sort instance indicates that the virtual role of undoability is imposed on all Commands\(^1\). The refactoring intention of the RSI sort is to move the implementation of this virtual role to an aspect using inter-type declarations, and optionally abstract the virtual role into a new interface—in this example the virtual role is indeed abstracted. The resulting aspect that implements this role superimposition is shown in Listing 5.3.

This aspect declares a newly created interface to be a parent of the Command interface. This new interface represents the role of undoability and is shown in Listing 5.4. The original Command interface no longer contains these methods, as shown in Listing 5.5. The aspect also makes all Commands implement the methods declared by the interface UndoableCommand. This is done with inter-type method declarations. It means that the Commands no longer implement these methods directly, as shown for the example Command in Listing 5.6.

\(^1\)This example is a modification and simplification of an actual RSI sort instance that is encountered in the case study in Chapter 7
5.1 Refactoring Intention of RSI Sort

Refactoring Role Superimposition Sort

```java
public aspect UndoableCommandRole {
    declare parents: Command extends UndoableCommand;

    // implementation of UndoableCommand for DeleteCommand:
    public Undoable DeleteCommand.getUndoActivity() {
        // ... [body] ... 
    }
    public void DeleteCommand.setUndoActivity(Undoable u) {
        // ... [body] ... 
    }
    public Undoable DeleteCommand.createUndoable() {
        // ... [body] ... 
    }
    // ... [implementation for other Commands] ...
}
```

Listing 5.3: Aspect implementation of virtual role superimposition

```java
public interface UndoableCommand {
    Undoable getUndoActivity();
    void setUndoActivity(Undoable u);
    Undoable createUndoable();
}
```

Listing 5.4: Newly created interface for undoable role

```java
public interface Command {
    void execute();
    // ... [more command-role methods] ...
}
```

Listing 5.5: Command interface no longer has undoable role

```java
public class DeleteCommand implements Command {
    public void execute() {
        // ... [body] ... 
    }
    // ... [methods not related to undoable] ...
}
```

Listing 5.6: Example Command without undoable related methods
5.2 Algorithm for Refactoring the RSI Sort

The generic algorithm described in Section 4.3 has some gaps that need to be filled in for the specific sorts. The RSI specific parts of the algorithm determine the required input, the refactoring problems and solutions, and the actual migration step. The complete algorithm with the sort-specific parts filled in for the RSI sort, provides a more concrete version of the intention described in the previous section.

5.2.1 Input to the algorithm

The input to the algorithm consists of an RSI sort instance and additional input from the developer. The sort instance provides the context code elements that participate in the refactoring and the developer provides the target aspect and possibly more information.

Input provided by sort

The RSI sort instance provides two types of elements. The first is a set of classes that have a role superimposed, the imposees. The second is a set of methods that make up the role that is being superimposed. This role can be virtual, if the set of methods does not make up one complete interface. The RSI sort specifies the relation between the context elements as that of a role and its imposees.

Input provided by developer

Besides providing the target aspect, the developer may need to indicate what to do with the role. If the role is not virtual, it can simply be imposed on the imposees using an inter-type parent declaration. If the role is virtual, however, only a subset of the methods in the role interface are part of the role and therefore imposing the complete interface on the imposees using an aspect would result in imposing more than the role.

If the role is virtual, an option is to abstract the role methods of the virtual role to a new interface. If the developer indicates this option, the name of the new role interface needs to be given. This new role will then be declared parent of the interface that originally contained the role methods. If the developer does not indicate this option, only the implementation of the methods of the virtual role will be moved to the aspect.

Input algorithm

The gathering of input for the RSI sort can be represented by a pseudo-code implementation of RSI specific version of the GATHERINPUT procedure in the generic Algorithm 4.1 on page 37. This RSI specific version of the algorithm of this procedure is shown in Algorithm 5.1.

In this algorithm we can see how the sort is added to the input (line 3). The developer is asked to provide a target aspect for the refactoring (line 4). If the role is virtual, the developer is also asked whether the virtual role should be abstracted to a new interface.
5.2 Algorithm for Refactoring the RSI Sort

Algorithm 5.1 Gathering input for the RSI sort

1: procedure GATHERINPUT(sort)
2:     input ← new INPUT
3:     input.sort ← sort
4:     input.targetAspect ← ASKDEVELOPER(“Target aspect…”)
5:     if sort.role.isVirtual then
6:         input.makeNewInterface ← ASKDEVELOPER(“Make new interface?”)
7:         if input.makeNewInterface then
8:             input.newName ← ASKDEVELOPER(“New role name…”)
9:         end if
10:     end if
11:     return input
12: end procedure

(line 6) and if so, to provide a name for this new role interface (line 8). The input provided by the developer is returned to be used in the other refactoring steps.

5.2.2 Possible Problems

The following is a list of problems that can occur in the context of refactoring an RSI sort, with a short description of how they occur. These problems can occur if the conditions for these problems are met, as outlined in Section 4.3.2. The result of the DETERMINEPROBLEMS procedure in Algorithm 4.1 on page 37 will consist of occurrences of these problems when refactoring an RSI sort instance.

Access of private method If an implementation of a role method in an imposee contains a call to a private method, moving the implementation to an inter-type method declaration will result in the usage of a private method from the aspect.

Access of private field If an implementation of a role method in an imposee contains a reference or assignment to a private field, moving the implementation to an inter-type method declaration will result in the usage of a private field from the aspect.

Access of package-visible method If an implementation of a role method in an imposee contains a call to a package-visible method, moving the implementation to an inter-type method declaration will result in the usage of a package-visible method from the aspect.

Access of package-visible field If an implementation of a role method in an imposee contains a reference or assignment to a package-visible field, moving the implementation to an inter-type method declaration will result in the usage of a package-visible field from the aspect.

Access of protected method If an implementation of a role method in an imposee contains a call to a protected method, moving the implementation to an inter-type method declaration will result in the usage of a protected method from the aspect.
Access of protected field If an implementation of a role method in an imposee contains a reference or assignment to a protected field, moving the implementation to an inter-type method declaration will result in the usage of a protected field from the aspect.

Access of a private inner class If an implementation of a role method in an imposee contains a reference to a private inner class, moving the implementation to an inter-type method declaration will result in the usage of a private inner class from the aspect. If one of the imposees is a private inner class, creating a declare parents and inter-type methods for that imposee will result in the usage of a private inner class from the aspect.

Access of a package-visible inner class If an implementation of a role method in an imposee contains a reference to a package-visible inner class, moving the implementation to an inter-type method declaration will result in the usage of a package-visible inner class from the aspect. If one of the imposees is a package-visible inner class, creating a declare parents and inter-type methods for that imposee will result in the usage of a package-visible inner class from the aspect.

Access of a protected inner class If an implementation of a role method in an imposee contains a reference to a protected inner class, moving the implementation to an inter-type method declaration will result in the usage of a protected inner class from the aspect. If one of the imposees is a protected inner class, creating a declare parents and inter-type methods for that imposee will result in the usage of a protected inner class from the aspect.

Access of a package-visible class If an implementation of a role method in an imposee contains a reference to a package-visible class, moving the implementation to an inter-type method declaration will result in the usage of a package-visible class from the aspect. If one of the imposees is a package-visible class, creating a declare parents and inter-type methods for that imposee will result in the usage of a package-visible class from the aspect.

Appendix B.1 has for each of these problems an example and the set of solutions that will resolve the problem, as well as a description of what these solutions will do in the context of refactoring an RSI sort instance.

5.2.3 Migration

The final step in Algorithm 4.1 on page 37 is calling the procedure PERFORMMigration (line 11), which is completely dependent on the sort instance being refactored. The implementation in pseudo-code of this procedure for the RSI sort is shown in Algorithm 5.2.

The first lines of Algorithm 5.2 (lines 2-4) consist of assigning information from the input and sort to local variables for ease of reference. After that comes a part that is dependent on whether or not the role is virtual (lines 5-21). The final part is independent of the role being virtual (lines 23-27) and takes care of migrating the implementation of the role methods from the imposees to the aspect.
Algorithm 5.2 Migrating RSI sort instances

1: procedure PERFORMMIGRATION(input)  
2:   sort ← input.sort  
3:   role ← sort.role  
4:   aspect ← input.targetAspect  
5:   if role.isVirtual then  
6:     if input.makeNewInterface then  
7:       newRole ← CREATENEWINTERFACE(input.newName)  
8:       for all role.methods as method do  
9:         ADDMETHOD(newRole, method)  
10:        REMOVEMETHOD(role, method)  
11:     end for  
12:     CREATEDECLAREPARENTS(aspect, role, newRole)  
13:   end if  
14:   else  
15:     for all sort.imposees as imposee do  
16:       if IMPLEMENTS(imposee, role) then  
17:         CREATEDECLAREPARENTS(aspect, imposee, role)  
18:         REMOVEROLEFROMCLASS(imposee)  
19:       end if  
20:     end for  
21:   end if  
22:  
23: for all sort.imposees as imposee do  
24:   for all role.methods as method do  
25:     MIGRATEINTERTYPEMETHOD(aspect, imposee, method)  
26:   end for  
27: end for  
28: end procedure

Virtual role: creating a new interface

If the role is virtual (line 5), the developer was asked in the GATHERINPUT procedure in Algorithm 5.1 whether a new interface should be created for this role. If the answer was NO, nothing is changed and the algorithm moves to the migration step on line 23.

If the answer was YES, the new role interface is created. This is done by first creating an empty role interface with the name specified by the developer (line 7). This new role is created in the same package as the original role. Next, each role method of the original role is added to the new role (line 9) and removed from the original role (line 10).

Finally, the new role is declared a parent of the original role, by creating a declare parents in the aspect (line 12). This will ensure that role methods called on the original role throughout the program will not be broken, now those methods are no longer in the original role, but in a new one.
Refactoring Role Superimposition Sort

5.2 Algorithm for Refactoring the RSI Sort

Algorithm 5.3 Migrating methods from imposee to aspect

1: procedure MIGRATE_TO_INTERTYPE_METHOD(aspect, imposee, rolemethod)
2: member ← GET_IMPOSEE_METHOD(imposee, rolemethod)
3: if method then
4: REMOVE_METHOD(imposee, method)
5: CREATE_INTERTYPE_METHOD(aspect, imposee, method)
6: end if
7: end procedure

Non-virtual role: declaring parents

If the role is not virtual (line 14), we can declare the role to be a parent of all the imposees in the aspect. For each imposee, we first check to see if they indeed implement the role (line 16). This can be FALSE if some imposee is a subclass of another imposee that does implement the role directly. This subclass then implements the role through its superclass, and no parent needs to be declared on this subclass.

If the imposee does implement the role directly, a declare parents is created in the aspect, that declares the imposee to implement the role (line 17). The role is then removed from the list of implemented interfaces in the imposee (line 18), since this is now done in the aspect.

Migration of role methods

Whether or not the role is virtual, the implementation of the role methods by the imposees will need to be migrated from the imposees to the aspect. In Algorithm 5.2 for each of the imposees (line 23), each method of the role (line 24) is migrated. This migration is done in the procedure MIGRATE_TO_INTERTYPE_METHOD (line 25). The pseudo-code implementation of this procedure is shown in Algorithm 5.3.

This procedure takes three arguments: the target aspect, the imposee whose role implementation to migrate and the method from the role that should be migrated. First, the actual implementation of this role method is obtained from the imposee (line 2). This implementation might not exist for the given imposee. If the imposee is a subclass of another imposee that does implement the current role method, the current imposee does not necessarily override the method.

If the imposee has an implementation of the role method (line 3), this implementation will be migrated to an inter-type method. First, the original implementation is removed from the imposee (line 4). Second, the implementation of the method is added to the aspect as an inter-type method for the imposee (line 5). This way the imposee still implements the role method, but the implementation is moved to the aspect along with all the other role related implementations.
5.3 Implementation

5.3.1 Relation of RSI refactoring with abstract SAIR classes

The user interface for the RSI sort is only specific in the input page, as can be seen in Figure 5.1. This figure shows how the default ResolveProblemsPage is used for displaying and resolving problems. The RoleSuperimpositionWizard is responsible for loading the RoleSuperimpositionInputPage page and displaying the proper name for the refactoring.

The AspectIntroducingRefactoring delegates the refactoring of the RSI sort to the RoleSuperimpositionDelegate, shown in Figure 5.2. This delegate uses the specific information from the sort instance and input pages to create the changes that represent the refactoring of the sort instance and that can be previewed in the wizard before being applied.
5.3 Implementation

```java
public interface OriginalInterface {
    void nonRoleMethod();
}

public interface NewInterface {
    void roleMethod(); // abstracted from OriginalInterface
}

public class ExampleImposee implements OriginalInterface,
                                NewInterface { // implements abstracted role directly
    public void nonRoleMethod() {
        // ... [body] ...
    }
    public void roleMethod() {
        // ... [body] ...
    }
}

public class SomeOtherClass {
    public void exampleMethod() {
        OriginalInterface orig = new ExampleImposee();
        orig.roleMethod(); // <= compile error
        orig.nonRoleMethod();
    }
}
```

Listing 5.7: Example of infeasible replacement of original by new interface

5.3.2 Design issues

Abstracting virtual roles

As was already mentioned in Section 5.1, when a virtual role is abstracted into a new interface, this new interface is declared parent of the originating interface, not of the imposees directly. Ideally it would be declared parent of all the imposees, but this is infeasible in strongly typed languages like Java, as the following example shows.

If we look at Listing 5.7, this is what would occur if we let the imposees implement the new interface directly. The method call on line 23 was valid before the new interface was created, since the method `roleMethod()` was a method of `OriginalInterface`.

However, after abstracting the role to `NewInterface`, this method is no longer in `OriginalInterface`. Although the variable `orig` is initialised using `ExampleImposee` (which does implement all methods), it is of type `OriginalInterface`, which does not declare `roleMethod()` anymore. This leads to the compile error on line 23.

Naively trying to solve this by replacing this type with `NewInterface` would create a new compile error on line 24. The feasible solution is to make the interface a parent of the original interface, which is what SAIR does.
5.4 Usage

Role interfaces and classes

Although the role in an RSI sort instance is simply a set of members belonging to the same interface or class, it was mentioned in Section 5.1 that SAIR can only handle the case in which the role is an interface and the role members are all methods. The reason that classes are not supported as roles is partly because of a misinterpretation of the RSI sort definition and partly because of a implementation simplification.

In the definition of the sort in [32], it says “… concrete instances [of the RSI sort] occur as multiple interfaces (or methods that can be abstracted into an interface) implementations.” (emphasis added). This was understood to mean that roles can only be made up of methods that are in an interface, or that can be abstracted into one.

Based on this understanding, the algorithm and implementation of SAIR were simplified by supporting only interfaces as roles (virtual or not) and only methods as (virtual) role members. This simplification was based on the observation of concrete instances of the RSI sort that all had only interfaces as (virtual) roles.

It is now understood that this interpretation is not complete. Classes can also be (virtual) roles, and fields can also be role members. However, the simplified algorithm for the refactoring of the RSI sort is sufficient for the case study. We leave the support of classes as roles and fields as role members as a possible future improvement.

5.4 Usage

The only part of the wizard that is unique for each sort—apart from the name—is the input page. This section shows how this input page is composed for the RSI sort.

5.4.1 Non-virtual role input

When the role of the RSI sort is not virtual, only the target aspect needs to be provided. This is shown in the wizard in Figure 5.3. This figure is similar to Figure 4.9, only this time it is explicit that only the target aspect is needed.
5.4 Usage

5.4.2 Virtual role input

If the role of the RSI sort is virtual, the developer may choose to abstract this virtual role to a new interface to better reflect the modularity of concerns. The input page for virtual role RSI sorts is shown in Figure 5.4.

Besides the target aspect, this input page lets the developer choose to abstract the virtual role to a new interface. To clarify which methods make up the virtual role, a list of these methods is presented. If the choice is to leave these methods in the current interface, the ‘Next’ button can be pressed (after the target aspect has been provided). If the developer chooses to abstract these methods to a new interface, a name for this new interface needs to be given, shown in Figure 5.5.
5.4.3 Further steps

The other steps of the refactoring are described in Section 4.5.3 for resolving problems and Section 4.5.4 for previewing the changes. These steps are general, regardless of the sort being refactored.

5.5 Summary

In this chapter, the algorithms details for the refactoring of the RSI sort were presented.

The input for the refactoring consists of at least the sort instance and the target aspect. If the role of the RSI sort instance is virtual, an option is to abstract this virtual role to a new dedicated interface. In this case, the name of the new interface is needed as additional input.

The migration step of the algorithm takes care of the migration of the role implementation to the target aspect. If the role is virtual, optionally the role methods are abstracted to a new interface. This interface is then declared parent of the original interface. If the role is not virtual, the role is declared a parent of all imposees. Finally, the implementation of the role methods for each of the imposees is migrated to inter-type method declarations, whether the role is virtual or not.

This chapter also presented some details of the implementation of the refactoring algorithm in SAIR. Finally, a short introduction to the input page of the RSI refactoring wizard was given, accompanied by screenshots.
Chapter 6

Refactoring Consistent Behavior Sort

Besides the RSI sort, the Consistent Behavior (CB) sort is implemented by SAIR. In this chapter we will see what the intention of refactoring a CB sort instance is. This intention describes how the refactoring will transform the code. The steps and details of the algorithm specific to the CB sort are also presented, as well as the implementation of these sort-specific steps. Finally, we show how refactoring the CB sort using SAIR can be done.

The CB sort is one of eleven other sorts (not including the RSI sort) identified by Marin in [36]. The CB sort, as explained in Section 2.4.2, describes crosscutting concerns that indicate the consistent invocation of an action by a set of methods to fulfil a requirement additional to their core functionality. This leads to tangling of this additional requirement with the primary functionality of the methods. It also leads to scattering, because the additional requirement is implemented in calls scattered among several methods. Concrete instances of the CB sort occur in object-oriented languages (such as Java) as a set of methods that consistently call one method that implements a common action.

6.1 Refactoring Intention of CB Sort

To understand the CB-specific details and steps of the algorithm, the intention of refactoring the CB sort should be clear. This intention indicates broadly what code transformation are needed to create an aspect-oriented implementation for CB sort instances. The intention will describe the input that is required and the resulting output. Such a broad overview will make it easier to understand the problems that need to be determined, and the algorithm steps.

The CB sort defines a method that is consistently called, the consistent method. It also defines a set of elements: the context methods in which the consistent method is called. This set of context methods is possibly related in a way other than all calling the consistent method. They could, for instance, all belong to the same hierarchy and consistently call a method on a common superclass. The method call to the consistent method is the consistent call.
6.1 Refactoring Intention of CB Sort

6.1.1 Refactoring CB to aspect

The desired aspect language mechanisms of the CB sort are pointcuts and advice [36]. Pointcuts are used to capture the context methods in which the consistent method is called. Using advice, these pointcuts can be advised by calling the consistent method, either before or after the context method captured by the pointcut.

The expression or statement containing the call to the consistently called method is the consistent statement. This is not necessarily a standalone method call in the context method. For instance, the method can be called on a field or on the return value of another method. When migrating the consistently called method from the context to advice, the complete consistent statement is moved, including the consistent call.

When refactoring an existing CB sort instance to an aspect, the pointcuts and advices need to be created for the contexts and the consistent method. This means that the consistent method is no longer called inside the context method, but in advice for those methods, in an aspect. The consistent statement needs to be moved from the context to an advice body, and pointcuts need to be created that capture the context so that it can be advised. This leads to an aspect solution of the CB sort.

6.1.2 Grouping advice or pointcut

There are three ways to create pointcuts and advice for the contexts and consistent calls. The easiest way to migrate the consistent calls from context methods to advice, is creating a unique pointcut for each context method, and advice that pointcut with its own advice containing the consistent statement. However, if the consistent statement is identical for some contexts and has the same location (first or last call in method), it is possible to group the pointcuts or the advice for those contexts in two ways.

One way of grouping is grouping by pointcut: creating one pointcut that captures all the context methods that have the same advice. Only one advice is required that advises this grouped pointcut.

The other way of grouping is grouping by advice: creating one advice for several pointcuts. This will still require one named pointcut for each context method but will group these pointcuts in an anonymous pointcut to advice the set.

6.1.3 Example refactoring

To further clarify the intention, the following example shows the before and after situation of refactoring a CB sort instance with pointcut grouping. Suppose there is an interface Command—the same interface as in the example for RSI in Section 5.1 on page 53—that is implemented by several different commands that are undoable.

At the end of their execute() method, these commands create an Undoable and set it as their undo activity by calling the method setUndoActivity(). This consistent behaviour is shown for two example Commands in Listing 6.1 and Listing 6.2.

The CB sort instance indicates that the setUndoActivity() method is consistently called in the execute() methods of all Commands. The refactoring intention of the CB
Refactoring Consistent Behavior Sort  

6.2 Algorithm for Refactoring the CB Sort

```java
public class DeleteCommand implements Command {
    public void execute() {
        // ... [execute code] ...
        setUndoActivity(createUndoable());
        // ... [other Command methods] ...
    }
}
```

Listing 6.1: Example DeleteCommand with consistent behaviour

```java
public class PasteCommand implements Command {
    public void execute() {
        // ... [execute code] ...
        setUndoActivity(createUndoable());
        // ... [other Command methods] ...
    }
}
```

Listing 6.2: Example PasteCommand with consistent behaviour

```java
public aspect UndoableCommandRole {
    pointcut commandExecute(Command c) :
        target(c) &&
        (within(DeleteCommand) ||
        // ... [other Commands] ...
        within(PasteCommand)) &&
        execution(public void execute());

    after(Command c) : commandExecute(c) {
        c.setUndoActivity(c.createUndoable());
    }
}
```

Listing 6.3: Aspect implementation of consistent behaviour

sort is to move the consistent calls to an aspect using pointcuts and advice. The resulting aspect that implements this consistent behaviour is shown in Listing 6.3.

This aspect has advice that consistently calls the `setUndoActivity()` after each `execute()` method called on a `Command`, in the same way it was done in at the end of the `execute()` methods. This means the consistent method is no longer called in the `execute()` methods themselves, as shown for an example `Command` in Listing 6.4.

### 6.2 Algorithm for Refactoring the CB Sort

The generic algorithm described in Section 4.3 has some gaps that need to be filled in for the specific sorts. In Section 5.2 these gaps are filled for the RSI sort. This section describes how these gaps can be filled for the CB sort.
6.2 Algorithm for Refactoring the CB Sort

The CB specific parts of the algorithm consist of—as is the case for the RSI sort—the input, the refactoring problems and the actual refactoring step. These steps are a more concrete version of the intention described in the previous section.

6.2.1 Input to the algorithm

The input to the algorithm consists of a CB sort instance and additional input from the developer. The sort instance provides the context code elements that participate in the refactoring and the developer provides the target aspect and some additional information.

Input provided by sort

The CB sort instance provides two types of elements. The first is the method that is called consistently in different contexts. The second is a set of methods that make up the context in which the method is consistently called. The CB sort specifies the relation between the context elements as that of a consistently called method and the context in which it is called.

Input provided by developer

Besides providing the target aspect, the developer may need to indicate whether to group the pointcuts or the advice. Depending on the answer, there will be no grouping, grouping by pointcut, or grouping by advice.

The developer can also indicate that instead of ‘after’ advice, ‘after returning’ advice should be used. The ‘after returning’ advice will then be created if the consistent call is last in the context. This prevents the advice from executing if the context method ends with throwing an exception instead of returning normally.

Input algorithm

The gathering of input for the CB sort can be represented by a pseudo-code implementation of CB specific version of the GATHERINPUT procedure in the generic Algorithm 4.1 on page 37. This CB specific version of the algorithm of the GATHERINPUT procedure is shown in Algorithm 6.1.

In this algorithm we can see how the sort is added to the input (line 2). The developer is asked to provide a target aspect for the refactoring (line 4). The developer is asked if there should be grouping (line 5) and if so, whether advice or pointcuts should be grouped (line
Algorithm 6.1 Gathering input for the CB sort

1: procedure GATHERINPUT(sort)
2:    input ← new INPUT
3:    input.sort ← sort
4:    input.targetAspect ← ASKDEVELOPER(“Target aspect…”)
5:    group ← ASKDEVELOPER(“Group advice/pointcuts?”)
6:    if group then
7:        grouping ← ASKDEVELOPER(“Group advice or pointcuts…”)
8:        if grouping = advice then
9:            input.groupAdvice ← TRUE
10:       else if grouping = pointcut then
11:            input.groupPointcuts ← TRUE
12:       end if
13:    end if
14:    input.afterReturning ← ASKDEVELOPER(“Use ‘after returning’?”)
15:    return input
16: end procedure

7). Finally, the developer is asked if ‘after returning’ advice should be used to advice the context if the consistent statement is last in the context, instead of ‘after’ (line 14).

6.2.2 Possible Problems

The following is a list of problems that can occur in the context of refactoring a CB sort, with a short description of how they occur. These problems can occur if the conditions for these problems are met, as outlined in Section 4.3.2. The result of the DETERMINEPROBLEMS procedure in Algorithm 4.1 on page 37 will consist of occurrences of these problems when refactoring a CB sort instance.

**Access of private method** If the consistent statement in a context contains a call to a private method, moving the statement to advice will result in the usage of a private method from the aspect.

**Access of private field** If the consistent statement in a context contains a reference to a private field, moving the statement to advice will result in the usage of a private field from the aspect.

**Access of package-visible method** If the consistent statement in a context contains a call to a package-visible method, moving the statement to advice will result in the usage of a package-visible method from the aspect.

**Access of package-visible field** If the consistent statement in a context contains a reference to a package-visible field, moving the statement to advice will result in the usage of a package-visible field from the aspect.
Access of protected method: If the consistent statement in a context contains a call to a protected method, moving the statement to advice will result in the usage of a protected method from the aspect.

Access of protected field: If the consistent statement in a context contains a reference to a protected field, moving the statement to advice will result in the usage of a protected field from the aspect.

Access of a private inner class: If the consistent statement in a context contains an explicit reference to a private inner class (e.g., by using a static field of the inner class), moving the statement to advice will result in the usage of a private inner class from the aspect.

Access of a package-visible inner class: If the consistent statement in a context contains an explicit reference to a package-visible inner class (e.g., by using a static field of the inner class), moving the statement to advice will result in the usage of a package-visible inner class from the aspect.

Access of a protected inner class: If the consistent statement in a context contains an explicit reference to a protected inner class (e.g., by using a static field of the inner class), moving the statement to advice will result in the usage of a protected inner class from the aspect.

Access of a package-visible class: If the consistent statement in a context contains an explicit reference to a package-visible class (e.g., by using a static field of the class), moving the statement to advice will result in the usage of a package-visible class from the aspect.

Access of super field: If the consistent statement in a context contains an explicit reference to a super field, moving the statement to advice will result in the usage of a super field from the advice.

Access of super method: If the consistent statement in a context contains an explicit call to a super method, moving the statement to advice will result in the usage of a super method from the advice.

Advising nested call: If the consistent statement in not in the direct body of the context method (but e.g., inside an if-block), the context cannot be advised with the consistent statement.

Advising neither first nor last call: If the consistent statement in not the first nor the last statement in the body of the context method, the context cannot be advised with the consistent statement.

Access of local variable: If the consistent statement uses a variable local to the context method, moving the statement to advice will result in the usage of a local variable from the advice.

For a list of solutions for each of these problem and a description of what these solutions will do in the context of CB, see Appendix B.2.
Algorithm 6.2 Migrating CB sort instances

1: procedure PERFORMMIGRATION(input)  
2: sort ← input.sort  
3: call ← sort.consistentCall  
4: aspect ← input.targetAspect  
5: contexts ← sort.contextMethods  
6: for all contexts as context do  
7:   pointcutinfos[] ← DETERMINEPOINTCUTINFO(context)  
8: end for  
9:  
10: if input.groupPointcuts then  
11:   grouped ← GROUPPOINTCUTS(pointcutinfos)  
12:   for all grouped as group do  
13:      CREATEGROUPEDPOINTCUT(group, aspect)  
14:      CREATEADVICE(group, aspect)  
15:   end for  
16: else if input.groupAdvice then  
17:   grouped ← GROUPPOINTCUTS(pointcutinfos)  
18:   for all grouped as group do  
19:     for all grouped.pointcutinfos as pointcutinfo do  
20:        CREATEPOINTCUT(group)  
21:     end for  
22:   end for  
23: else  
24:   for all pointcutinfos as pointcutinfo do  
25:     CREATEPOINTCUT(pointcutinfo, aspect)  
26:     CREATEADVICE(pointcutinfo, aspect)  
27:   end for  
28: end if  
29:  
30: for all contexts as context do  
31:   REMOVECALLFROMCONTEXT(context, call)  
32: end for  
33: end procedure  

6.2.3 Migration

The final step in Algorithm 4.1 on page 37 is calling the procedure PERFORMMIGRATION (line 11), which is completely dependent on the sort instance being refactored. The implementation in pseudo-code of this procedure for the CB sort is shown in Algorithm 6.2.

The first lines of Algorithm 6.2 (lines 2-5) consist of assigning information from the input and sort to local variables for ease of reference. The rest of the algorithm takes care of
the actual migration—creating the pointcuts and advice and removing the original call from the context.

**Determining pointcuts**

The first step in creating pointcuts and advice, is determining what the pointcuts are. This is done in the algorithm on lines 6-8. For all the context methods, information on the pointcut is determined by calling `DETERMINEPOINTCUTINFO` and added to the list `pointcutinfos`.

The `pointcutinfo` created by the `DETERMINEPOINTCUTINFO` procedure contains information of the context, including a pointcut definition. This pointcut matches only the execution of the given context method. Advice created for this pointcut will only apply before or after that method is executed. Other information in the pointcutinfo is the whether to use `before` or `after` advice (depending on the location of the consistent statement in the context method) and what the exact consistent statement is that needs to be put in the advice.

**Grouping**

Once all pointcut information is determined, the pointcut and advice constructs need to be added to the aspect. As explained in the refactoring intention in Section 6.1, pointcuts or advice can be grouped if the developer indicates so.

**Grouping by pointcut (lines 11-15)**

Grouping by pointcut creates a grouped pointcut for contexts that can be advised using the same advice. Grouping of pointcutinfos is determined by calling `GROUPPOINTCUTS` (line 11). The exact algorithm for grouping is not straightforward and quite implementation dependent—see Section 6.3 for how this is done in SAIR.

When the grouped pointcuts are determined, a pointcut is created in the target aspect for each group (line 13) as well as one advice for each group (line 14).

**Grouping by advice (lines 17-23)**

Grouping by advice creates one pointcut for each context method, but one grouped advice for all pointcuts that could have been grouped. Again grouped pointcutinfos are determined by calling `GROUPPOINTCUTS` (line 17).

First for each group, a pointcut is created for each of the pointcuts in the group (lines 19-21). Next one advice is created for the whole group (line 22).

**No grouping (lines 25-28)**

The last option for grouping is no grouping. This simply creates one pointcut in the target aspect for each pointcutinfo (line 26) and one advice for each pointcut (line 27).

**Removal of consistent statement**

Finally, the original consistent calls need to be removed from the context methods. This is done on lines 31-33 in the algorithm. For each context method, `REMOVECALLFROMCONTEXT` is called which removes the consistent call from the given context method. Of course
Figure 6.1: Simplified class diagram showing CB specific wizard classes.

this will not modify the behaviour of the program, since the method is now consistently called in the appropriate advice.

6.3 Implementation

6.3.1 Relation of CB refactoring with abstract SAIR classes

The user interface for the CB sort is only specific in the input page, as can be seen in Figure 6.1. This figure shows how the default ResolveProblemsPage is used for displaying and resolving problems. The ConsistentBehaviorWizard is responsible for loading the ConsistentBehaviorInputPage and displaying the proper name for the refactoring.

The AspectIntroducingRefactoring delegates the refactoring of the CB sort to the ConsistentBehaviorDelegate, shown in Figure 6.2. This delegate uses the specific information from the sort instance and input pages to create the changes that represent the refactoring of the sort instance and that can be previewed in the wizard before being applied.

6.3.2 Design issues

Consistent statement

As mentioned before in Section 6.1.1, the consistent call need not be a standalone method invocation. It can be part of a larger statement in the method body, the consistent statement. Since only complete statements can be moved from the method body to the advice body, we need to locate this statement. Listing 6.5 shows such a consistent statement, where the method invocation is on a field.

Locating this consistent statement is done using the AST of the context class and the concept of visitors. A visitor is a class that can be used to recursively traverse a tree structure, such as the AST. The visitor has a visit method for each type of AST node so for each type of node a specific action can be performed.

We visit only the part of the AST that is the context method. As soon as the visitor encounters a method invocation, the binding is checked. If the invoked method is indeed the
6.3 Implementation

Figure 6.2: Simplified class diagram showing CB specific refactoring classes.

```java
public class ExampleClass {
    private SomeClass field;
    public void exampleContext() {
        // ... [method body] ...
        field.consistentMethod();
    }
}
```

Listing 6.5: Example of consistent method call in context

consistent method or an overridden version of the consistent method, a flag is set, signalling that the visitor is currently inside the consistent statement. When ending the visit, the tree is tracked back towards the root. The first statement encountered when backtracking while the inside flag is set, is the consistent statement.

Statement information

Because the consistent statement is not necessarily a single method invocation, additional information may be needed in the advice varies depending on the concrete statement. For example, the class enclosing the context method is usually needed to call the method on from the advice.

Again a visitor is used to traverse the context method subtree to determine what kind of elements are needed in the advice. These elements also need to be captured by the pointcut. The possible elements are the class enclosing the context method and method parameters. An example pointcut and advice that capture and use the enclosing class, are shown in Listing 6.6.
The visitor is also used to determine problems. When a local variable is encountered in the consistent statement, a ‘Access of local variable’ problem is determined. When a field, method or class is encountered in the statement, its visibility is checked and problems are created if it is not visible from the target advice.

Finally, when the consistent statement is moved to advice, any fields and methods used in the statement need to be qualified with the captured surrounding type. The result of this can also be seen in Listing 6.6, where \( cc \) is put before the method call and the field used as parameter. Again a visitor is used to locate all the fields and methods in a consistent statement and qualify them with the correct context.

### Grouping

Grouping the contexts in one pointcut if they can be advised with the same advice is a very powerful feature that is not straightforwardly implemented. The problem lies in the fact that the consistent statements in the contexts cannot simply be compared character by character. This would easily result in incorrect grouping where pointcuts are grouped while they cannot be, or in too little grouping when pointcuts are not grouped while they can be.

An example of where simple string comparison is not enough, is shown in Listing 6.7. Both consistent statements are the same: the same `consistentMethod()` is called. But if we look at how these could be advised, we see that the statement 1 uses the argument of the method, and statement 2 uses a field. This means that we cannot use the same advice and pointcut for both context methods.

Once again, we use a visitor to solve this problem. The visitor traverses the consistent statement and resolves all bindings. Each binding is used to create a unique id for the consistent statement. If two consistent statements have the same id, they are indeed the same. In the example, the ids of statement 1 and 2 would not be identical, because the binding of `index` is not the same (method parameter in statement 1 and field in statement 2).

If the consistent statement ids are identical this means that they can be grouped, but it might still need to be necessary to find a common superclass of the different context methods to be able to create the correct parameters for advice and pointcut definitions.
6.4 Usage

The only part of the wizard that is unique for each sort—apart from the name—is the input page. This section shows how this input page is composed for the CB sort.

First of all, the target aspect needs to be provided since that is the input all refactorings need. This is shown in Figure 6.3 in the top section of the wizard.

Listing 6.7: Example of consistent method call in context

```java
public class ExampleClass1 extends CommonSuper {
    public void exampleContext(int index) {
        // ... [method body] ...
        consistentMethod(index); // <= consistent statement 1
    }
}

public class ExampleClass2 extends CommonSuper {
    private int index;
    public void exampleContext(int i) {
        // ... [method body] ...
        consistentMethod(index); // <= consistent statement 2
    }
}
```

Figure 6.3: Providing input for the CB refactoring
6.4.1 Input specific to CB sort refactoring

Figure 6.3 also shows how grouping of pointcuts can be indicated in the middle section of the wizard. The default behaviour is no grouping, which means one pointcut and one advice is created for each context method.

Optionally, a pointcut template can be provided that will be prepended to the pointcuts created in this refactoring. The template will be prepended with an underscore (_) as glue to the rest of the pointcut name that will be determined by the context(s) for which the pointcut is made. This means this template cannot contain characters that cannot be in an AspectJ pointcut name.

Additionally, the developer can indicate that contexts where the original consistent statements have been removed should be tagged with a special SAIR comment. This comment makes it easier to later identify the elements that have been removed.

Finally, the developer can indicate whether after returning advice should be used instead of after advice where appropriate. This option is checked by default because unless the consistent call is inside a catch or finally block, the original call would only have been executed if the method returned normally and this is the behaviour captured by the after returning advice.

6.4.2 Further steps

The other steps of the refactoring are described in Section 4.5.3 for resolving problems and Section 4.5.4 for previewing the changes. These steps are general, regardless of the sort being refactored.

6.5 Summary

In this chapter, the algorithms details for the refactoring of the RSI sort were presented.

The input for the refactoring consists of at least the sort instance and the target aspect. Additionally, grouping of pointcuts or advice may be requested, creating combined pointcuts and advice when several context methods can be advised with the same advice. Optionally, the use of after returning advice instead of after advice can be requested.

This migration step of the algorithm takes care of the migration of the consistent calls to the target aspect. If grouping was requested in the input, the pointcuts and advice are grouped accordingly. The resulting pointcuts and advice are created, with the advice containing the consistent statement for the pointcut that captures the original context method. Finally, the original consistent call is removed from the context methods.

This chapter also presented some details of the implementation of the refactoring algorithm in SAIR. Finally, a short introduction to the input page of the CB refactoring wizard was given, accompanied by a screenshot.
Chapter 7

Case Study - JHOTDRAW

The previous three chapters described SAIRe—the tool that is able to refactor RSI and CB sort instances to aspects—and the underlying algorithm. In this chapter, we will perform a case study with SAIRe, by applying the tool to an AspectJ benchmark project: JHOTDRAW. We will first present JHOTDRAW and the concern model we used in this case study. Next, we will describe the process of refactoring the concern model to aspects. Finally, we will discuss the process and the results of the case study.

7.1 JHOTDRAW

JHOTDRAW is an open source Java GUI framework for technical and structured two-dimensional graphics, originally developed by Gamma and Eggenschwiler as a ‘design exercise’ [18]. Its design is based on many well-known GoF\(^1\) design patterns [19].

There is an aspect-oriented refactoring of JHOTDRAW called AJHOTDRAW[38]. Because JHOTDRAW is regarded as a properly designed program, it is an ideal candidate to function as a showcase for aspect-oriented implementations. AJHOTDRAW is a manual refactoring of the concerns in JHOTDRAW to aspects. We will compare those manually created aspects with the aspects created by SAIRe.

The current version of JHOTDRAW is 7.0.8. However, since AJHOTDRAW is based on version 6.0b1, we will be using this older version to be able to better compare the results of our approach to the manually created aspects of AJHOTDRAW.

An overview of some relevant classes for this case study is shown in Figure 7.1. We will mainly be working on the Command hierarchy, but the other classes give a short overview of how JHOTDRAW is composed. The application has a DrawingWindow which has a DrawingView. On this view a Drawing is display that contains Figures.

Commands are classes that implement the Command interface. These commands modify the drawing and the view and implement the ‘Command’ design pattern [19]. Commands are undoable and the UndoCommand is responsible for executing the undo activity of an undoable command.

\(^1\)Gang of Four, so called because of the four authors of the original design pattern book.
7.2 Concern Model

The concern model is based on the concern model that can be found on the SoQUET website\(^2\). The model found there is for JHOTDRAW 5.4b1, but the only difference between that version and version 6.0b1 used in this case study, is the package names. The packages in version 5.4b1 start with `CH.ifa.draw` and have been renamed to `org.jhotdraw` in version 6.0b1.

The concern model was initially built in the normal way, but then converted to a SAIR compatible model, as discussed in Section 4.4.4. It is the normal version we will be presenting here, since the SAIR variant is not really different, but less conceptually related to the concrete concerns. E.g. talking about the hierarchy of some class (original model) shows more relation than a list of all the separate classes (SAIR version).

The model we use contains the CB and RSI concerns related to commands only—other concerns are not considered in this case study. The concerns are grouped under the ‘Command’ composite sort. There are two further groupings: observer related concerns and undo related concerns. Some general command sort instances are directly in the ‘Command’ composite sort. This concern model is shown in Figure 7.2 without details. The details of the classes and methods involved in the concern model are given in Appendix C.

We will now briefly explain what each of the sort instances in the ‘Command’ concern model means, before we start to refactor the concerns to aspects.

7.2.1 Main command concerns

These are the main concerns related to commands in JHOTDRAW in general.

**Command-MainRole (RSI)** A virtual role that indicates the four main role methods in the `Command` interface. All classes in the project that implement the `Command` interface have this role superimposed.

---

\(^2\)http://swerl.tudelft.nl/twiki/pub/AMR/SoQueT
7.2 Concern Model

InitCommand (CB) Indicates the consistent initialisation of commands by calling the constructor of AbstractCommand. Captures all constructors of classes in the hierarchy of Command that invoke this constructor.

PreExecutionCheck-ViewNotNull (CB) Indicates the consistent checking before execution, done in the execute() method of AbstractCommand. Captures all methods in the hierarchy of Command that call the execute() method of the class AbstractCommand.

PostExecutionNotification-UpdateToCmdChanges (CB) Indicates the consistent notification after execution. Captures all methods in the hierarchy of Command that call the checkDamage() method on the view.

7.2.2 Undoable command concerns

These are concerns related to the undoability of commands.

CommandUndoableRole (RSI) A virtual role that indicates two undo-related methods in the Command interface. All classes in the project that implement the Command interface have this role superimposed.

InitUndoSupport (CB) Indicates the consistent initialisation of the undo functionality in commands. Captures all methods in the hierarchy of Command that call the method setUndoActivity() of the Command interface.

SaveFiguresStateBeforeExecution (CB) Indicates the consistent saving of figures for undo before execution commands. Captures all methods in the hierarchy of Command that call the setAffectedFigures() method on their undo activity.

7.2.3 Observer command concerns

These are concerns related to the ‘Observer’ pattern the commands are involved in.
7.3 Refactoring

ObservableCommand (RSI) A virtual role that indicates two observable-related methods in the Command interface. All classes in the project that implement the Command interface have this role superimposed.

CommandListener (RSI) A role that indicates classes that will register as listeners to commands. All classes in the project that implement the CommandListener interface have this role superimposed.

7.3 Refactoring

With the concern model in place, we are now ready to refactor all the CB and RSI sort instances to aspects. This section explains the process of refactoring for the case study.

7.3.1 Setting up Eclipse

Since SAIR and SOQUET are Eclipse plug-ins, we need for this case study an Eclipse distribution with those plug-ins installed. This automatically requires (for SAIR) the AJDT plug-in as well. Of course the JDT and LTK also need to be present. The relevant versions used in this case study are:

- Eclipse (SDK): 3.2.2 (includes JDT 3.2.2)
- AJDT: 1.4.2
- AspectJ: 1.5.4 (included in AJDT)
- SOQUET: 0.2.0

A Java project should be present for JHotDraw and called ‘JHotDraw 6.0b1’. The concern model is expecting a project with that exact name. The main source folder is called ‘src’ and contains all the org.jhotdraw packages. Before refactoring, this project should be converted to an AspectJ project. This can be done by right-clicking on the project and selecting ‘AspectJ Tools’ in the context menu. There is only one option: ‘Convert to AspectJ Project’.

Because the concerns in the concern model are spread over different packages, it is not directly clear where the target aspects should be. This problem is solved by creating a package org.jhotdraw.ccconcerns, inspired by a similar package in the AJHotDraw project. Different concerns will have their own sub-package in this package.

7.3.2 Importance of refactoring order

The order in which the various concerns are refactored can be important. First, SAIR will not be able to locate methods that are inside aspects since the JDT is used for finding the methods. This makes it impossible to refactor consistent calls inside a role method after this

---

3 SAIR uses a custom version of SOQUET (which was built on request) that exposes all packages to SAIR
method has been migrated from a class to an aspect. This means that first the CB refactoring should be performed and then the RSI, when these sort instances might overlap.

Second, although this does not appear in the concern model for this case study, a role in an RSI concern might not be labelled ‘virtual’, but some other RSI instances may use the same interface and do label some methods virtual. When these virtual roles are refactored, the interface used by the non-virtual role is getting smaller when the virtual methods are abstracted to a new interface. When the non-virtual role is migrated to the aspect first, it will migrate all methods, including those of virtual roles, if those virtual roles are not migrated first.

Finally, since several different CB sort instance may be competing to be the first or last call in a method, the order in which those instance are refactored is important. First, the CB sort instance that actually is the first or last call in a method, the order in which those instance are refactored is important. First, the CB sort instance that actually is the first or last call in methods should be refactored. The second or penultimate call now becomes the first or last call in that method, respectively, and can be easily refactored as such. Of course care should be taken with precedence of advice if the order of these calls matters.

7.3.3 Refactoring main command concerns

The first step is that of refactoring the main command concerns. A quick scan through the different concerns in the concern model confirms that CB sort instances related to the main command concerns are indeed called first and last in most context methods and should therefore be refactored first.

InitCommand (CB)

The ‘InitCommand’ sort instance seems a good choice to start the refactoring with. The name suggests an initialisation step, something that is done first thing in commands. In fact, it indicates that all Commands consistently invoke the constructor of AbstractCommand as an initialisation step.

However, this sort instance cannot be refactored by SAI. The problem with migrating such a consistent call to an aspect, is that AspectJ cannot invoke constructors from advice. Since this is exactly what is needed in the aspect solution for this concern, it cannot be refactored. All consistent constructor invocations have this problem, but this is the only occurrence in this concern model.

PreExecutionCheck-ViewNotNull (CB)

This sort instance captures the consistent calling of the execute() method in the AbstractCommand. In this method, the view is checked for not being null, hence the name of the concern. A typical context method in which this is called is shown in Listing 7.1.

The target aspect used is the newly created CommandConsistency in the package org.jhotdraw.ccconcerns.commands. The grouping is set to ‘pointcut’ grouping, since the captured contexts are very similar and can therefore easily be grouped. As an optional pointcut name template, ‘viewNotNullContext’ is set and finally the original
locations should be marked. Since before advice is likely to be created (the calls are first in the methods), the ‘after returning’ option is not relevant.

There are 17 consistent call contexts and for all 17 of them the same refactoring problem is detected: ‘Access of super method’. This makes sense since the consistent call is super.execute() on AbstractCommand and super methods cannot be called from advice. One of the solutions to this problem is COPY_SUPER_BODY_TO_ADVICE, which will replace the call with the actual body of the super method. This solutions is applied for all problems.

When the refactoring is performed, a grouped pointcut is created that captures all contexts and is advised using before advice containing the body of the execute() method in AbstractCommand. The result is that the consistent call is removed from the context, as shown in Figure 7.2.

PostExecutionNotification-UpdateToCmdChanges (CB)

This sort instance captures the consistent calling of the checkDamage() method on the view associated with a command. In this method, the view checks for ‘damage’ and updates when appropriate. A typical context method in which the consistent method is called is shown in Listing 7.3.

The target aspect is again CommandConsistency since this concern is also a con-
### 7.3 Refactoring

Consistency concern of commands. Grouping is set to ‘pointcut’, since the consistent statement is very similar for most context methods. The template name given for the pointcuts is ‘postExecutionCheck’. We request to use after returning instead of after advice which is relevant because this check is last in the `execute()` method and will probably generate after advice.

With this concern however, several interesting problems are determined. The first is an ‘Advising nested call’ problem that occurs in `PasteCommand`. The situation is shown in Listing 7.4. It can be seen that the consistent call (`checkDamage()`) is last in the method, but inside an if block. However, if the check would be outside the if block, it would simply find there is no damage if `selection` was null. We therefore select the solution `ADVICE AS AFTER`, that will create after advice (or actually after returning advice) for this context.

The other problems concern the `UndoCommand` and the `RedoCommand` classes. Since these classes actually execute `Undoables` from other commands, their check is a little different. This is shown for `UndoCommand` in Listing 7.5.

This listing shows how two problems occur. First, the consistent call is not the first or last in the method, but second-to-last. Second, the consistent statement contains a reference to a local variable, `lastUndoable`, which cannot be accessed from advice if the call would be migrated. A similar situation occurs in the `RedoCommand` class.

Possible solutions for these problems are to simply advise using after advice and

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice in aspect ...
    setUndoActivity(createUndoActivity());
    getUndoActivity().setAffectedFigures(view().selection());
    ((GroupCommand.UndoActivity)getUndoActivity()).groupFigures();
    view().checkDamage(); // <= consistent statement
}
```

Listing 7.3: Consistent call for ‘PostExecutionNotification’ concern in `GroupCommand`

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice...
    Point lastClick = view().lastClick();
    FigureSelection selection = (FigureSelection)Clipboard
        .getClipboard().getContents();
    if (selection != null) {
        setUndoActivity(createUndoActivity());
        getUndoActivity().setAffectedFigures(
            (FigureEnumerator)selection.getData(
                StandardFigureSelection.TYPE));
        // ... [other code] ...
        view().checkDamage(); // <= consistent statement
    }
}
```

Listing 7.4: Nested consistent call problem in `PasteCommand`
7.3 Refactoring

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice ...
    UndoManager um = getDrawingEditor().getUndoManager();
    // ... [other code] ...
    Undoable lastUndoable = um.popUndo();
    // ... [other code] ...
    lastUndoable.getDrawingView().checkDamage(); // <= consistent stat.
    getDrawingEditor().figureSelectionChanged(lastUndoable.getDrawingView());
}
```

Listing 7.5: Consistent call of `checkDamage()` in `UndoCommand`

make the reference local `lastUndoable` a field. However, this case does not really fit the consistent behaviour of the other commands. It will require some further attention before it can be properly refactored. Therefore, the `UndoCommand` and `RedoCommand` will be excluded from this refactoring.

**Command-MainRole (RSI)**

The ‘Command-MainRole’ sort instance indicates the main virtual command role that all commands have. This is the `Command` interface without the methods of the virtual roles of ‘ObservableCommand’ and ‘CommandUndoableRole’.

Although this sort instance indicates a role, it has not been refactored for this case study. Since it indicates the main role of commands, and not a secondary role, it is not useful to migrate these methods to an aspect. All classes that implement the `Command` interface are, after all, commands.

### 7.3.4 Refactoring undo-related command concerns

Now the main roles have been refactored, we can start with the undo concern. First the CB sort instances will be refactored, then the RSI sort instance for the virtual role of undoability.

**InitUndoSupport (CB)**

In the `execute()` method of `Commands`, the first call is now the consistent call of the ‘Init-UndoSupport’ sort instance, as can be seen in the representative example Listing 7.6. This is the next sort instance under refactoring.

This concern is refactored to a newly created aspect `CommandUndoConsistency` in the package `org.jhotdraw.ccconcerns.commands.undo`. Because the creation of undo activities is different for each `Command`, we do not request grouped pointcuts since they cannot be grouped anyway. The pointcut template used is ‘initUndo’ and we request to leave a comment in the contexts where the calls have been removed.

There are thirteen problems determined, which at first sight seems a lot. Most of these problems however—twelve—are very similar, namely ‘Access of protected method’ prob-
7.3 Refactoring

Listing 7.6: Consistent call for ‘InitUndoSupport’ in GroupCommand

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice in aspect ...
    setUndoActivity(createUndoActivity()); // <= consistent statement
    getUndoActivity().setAffectedFigures(view().selection());
    ((GroupCommand.UndoActivity)getUndoActivity()).groupFigures();
    // @SAIR: view().checkDamage(); moved to advice in aspect ...
}
```

Listing 7.7: Nested and duplicate consistent call in PasteCommand

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice in aspect ...
    Point lastClick = view().lastClick();
    FigureSelection selection = (FigureSelection)Clipboard.getClipboard().getContents();
    getClipboard().getContents();
    if (selection != null) {
        setUndoActivity(createUndoActivity()); // <= consistent statement
        getUndoActivity().setAffectedFigures(
            (FigureEnumerator)selection.getData(StandardFigureSelection.TYPE));
        if (!getUndoActivity().getAffectedFigures().hasNextFigure()){
            setUndoActivity(null);
            return;
        }
    } // ... [more code] ... 
    // @SAIR: view().checkDamage(); moved to advice in aspect ...
}
```

Listing 7.6 shows how in the consistent statement the method `createUndoActivity()` is called, which is protected. Most `Command`s implement such a method, although it is not enforced through any interface. These problems are resolved by making the aspect privileged.

The last problem is a ‘Nested call’ problem, again in `PasteCommand`. This time it is not immediately clear if `before` advice can be used like it was possible to use `after` advice in the ‘PostExecutionCheck-updateToCmdChanges’ sort instance. This is because the call is not only nested, it is not the first call in the method and may depend on the previous calls, as can be seen in Listing 7.7. In fact, an undo activity should only be set if the command is indeed executed, which is only the case inside the `if` block.

Additionally, there is a peculiarity with the `PasteCommand` context method: the consistent method is called twice in this `execute()` method. The second call is part of a check that will cancel execution and resets the undo activity initialised in the first call. This is visible on line 13 in Listing 7.7.

These issues are reasons to exclude this context method from this refactoring, with the intention to look at them later. The ‘Nested call’ problem is therefore resolved with the
7.3 Refactoring

Case Study - JHotDraw

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice in aspect ...
    // @SAIR: setUndoActivity(createUndoActivity()); moved to ad...
    getUndoActivity().setAffectedFigures(view().selection());
    // ----------------------------- <= consistent call
    (GroupCommand.UndoActivity)getUndoActivity().groupFigures();
    // @SAIR: view().checkDamage(); moved to advice in aspect ...
}
```

Listing 7.8: Consistent call for ‘SaveFigureState’ concern in GroupCommand

EXCLUDE_METHOD solution.

SaveFiguresStateBeforeExecution (CB)

In the execute() methods of most Commands, the first two calls have migrated and the next call is the consistent call of the ‘SaveFigureStateBeforeExecution’ sort instance. This is shown for the representative GroupCommand in Listing 7.8, the consistent call is to the method setAffectedFigures. This is the sort instance we will refactor next.

Since this concern is also related to the consistent initialisation of undo in commands, the target aspect is the existing CommandUndoConsistency. The grouping is set to ‘pointcut’ and the pointcut template is ‘saveFigureState’. We also request comments to be created in the original context. Since this is not the last call in any context, the after returning option is not relevant.

Eight problems are detected—four for inadvisable contexts because of the location of the consistent call and four for the local variables used in those calls. The problems relate to DuplicateCommand, CutCommand, DeleteCommand and PasteCommand. In the first three cases, the consistent call is simply not the first, because some additional steps are needed to gather the figures whose states need to be saved. In the last case, PasteCommand, the consistent call is not only not the first, but also nested inside an if block.

Additionally, the consistent call is made twice in the execute() methods of DuplicateCommand and PasteCommand. Because this complicates the refactoring too much for now, these two classes are excluded from the refactoring by selecting the EXCLUDE_METHOD solution for the problems. The problem list is automatically refreshed and the local variable problems for those two contexts no longer appear.

Now we look at the similar problems of CutCommand and DeleteCommand where the consistent statement is not the first in the method and therefore not advisable, as shown for DeleteCommand in Listing 7.9. It can be seen that all the code prior to the consistent statement is also related to this consistent statement. The same can be observed in the CutCommand. We therefore select the ADVISE_AS_BEFORE solution for these problems. After refactoring, we can manually move the code preceding the consistent call to the respective advice. The local variable problems—the use of fe in the consistent statement—are ignored, because they will be fixed with the manual refactoring afterwards.
Case Study - JHotDraw

7.3 Refactoring

```java
public void execute() {
    // @SAIR: super.execute(); moved to advice ...
    // @SAIR: setUndoActivity(createUndoActivity()); moved ...
    // ricardo_padilha: bugfix for correct delete/undelete behavior
    /* When enumerating the affected figures we must not forget the
    * dependent figures, since they are deleted as well!
    */
    FigureEnumeration fe = view().selection();
    List affected = CollectionsFactory.current().createList();
    // ... [collecting affected figures] ...
    fe = new FigureEnumerator(affected);
    getUndoActivity().setAffectedFigures(fe);
    /* ricardo_padilha: end of bugfix */
    deleteFigures(getUndoActivity().getAffectedFigures());
    // @SAIR: view().checkDamage(); moved to advice ...
}
```

Listing 7.9: Consistent call with preceding related code in DeleteCommand

CommandUndoableRole (RSI)

The ‘CommandUndoableRole’ sort instance indicates the virtual undoable role that all Commands have. This virtual role consists of two methods: setUndoActivity and getUndoActivity. We indicate in the refactoring that these should be abstracted to a new interface called CommandUndo. The target aspect is the newly created Command-UndoRole in the org.jhotdraw.ccconcerns.commands.undo package.

The context of this sort instance is the hierarchy of Command. There are only two imposees that actually implement these methods: AbstractCommand and UndoableCommand—the other classes inherit these implementations.

There are two problems determined, both concerning the use of a private field myUndoableActivity. This is a field only related to the role. We choose to ignore these problems and manually refactor this field to the aspect after the current refactoring.

7.3.5 Refactoring observer-related concerns

The final two refactorings are related to the observer pattern implementation of which the commands are part.

ObservableCommand (RSI)

The ‘ObservableCommand’ concern indicates that a virtual role of two methods of the Command interface is superimposed on classes in the hierarchy. The two methods are related to the registering and unregistering of listeners to commands.

The implementation of this role is refactored to a newly created aspect Command-Observer in the package org.jhotdraw.ccconcerns.commands. The virtual role is abstracted to a new interface called ObservableCommand.
There are four problems determined, all ‘Access of protected method’ problems that try to access two (different) methods called `getEventDispatcher()` for (un)registering listeners. All these problems are resolved by making the methods public, because eventually this method will probably be refactored to aspects as well as it indicates the secondary concern (not captured by this role) of dispatching the events.

CommandListener (RSI)

The ‘CommandListener’ role indicates that all classes implementing the interface `CommandListener` have a secondary role superimposed. This role is not virtual, so the complete interface is the role and can be declared parent of all imposees directly in the aspect.

Since this role is also related to the observability concern, we refactor this role to the `CommandObserver` as well. No problems were detected, but we did notice the used of the previously protected method `getEventDispatcher()`, which goes to show that problem detection is sensitive to the order of refactoring.

7.4 Discussion

7.4.1 Refactoring process

This case study was deliberately performed without manual object-oriented refactoring up front, to be able to assess the functionality of `SAIR` on a normal, somewhat sloppy code base. This created small problems with the refactoring of CB sort instances concerning the creation of pointcuts and advice.

One class that was particularly hard to refactor, was the `PasteCommand` class. The tangled nature of the undo concern in the `execute` method of this class was shown in Listing 7.7. Although this tangling of the undo concern with the primary execution functionality does indicate the problem with CCCs quite nicely, it makes it very hard to refactor this concern for the `PasteCommand` to the aspect.

The `PasteCommand` class could have benefited from object-oriented pre-refactoring. Thorough refactoring might be able to fix the problem with the location of the consistent call, possibly by moving some of the code to separate (private) methods. The problems of making the consistent call twice in the method should also be refactored out before attempting to refactor the complete concern to an aspect.

On two occasions there was some additional manual refactoring after refactoring a concern. The first was with the ‘SaveFiguresStateBeforeExecution’, where some additional code was moved to advice after refactoring because it was related to the consistent call. The second occasion was when the field `myUndoActivity` was moved to the aspect along with the role of undoability, because only that role used the field.

This is something that can be expected to happen. Even with normal Eclipse refactorings, the result is not always exactly as desired and some manual modifications or additions to the refactoring result are made. `SAIR` cannot be prepared for all situations, although
7.4 Discussion

some more refactoring problems can be imagined with more solutions to solve some of these problems, as we will discuss in Section 8.2.

A concern that could not be refactored at all, was the ‘InitCommand’ CB sort instance. The problem is that AspectJ cannot invoke constructors from advice, something that is needed for the straightforward aspect solution. A workaround could be to move the content of the consistently invoked constructor to a separate method, and invoke that method from advice.

7.4.2 Refactoring result

The aspects created by the refactoring are vary in quality for different sort instances. Some concerns could be refactored to sorts relatively easily and created nice aspects. Other concerns where more problematic with a higher degree of tangling, creating less nice aspects.

An example of a nicely created aspect is shown in Listing 7.10 as a result of refactoring the ‘UndoableCommand’ role. The inter-type field on line 5 was later added with a manual refactoring but that is only a small change. This aspect clearly implements the role defined by the sort instance.

Another nice aspect is shown for the main command consistency concerns in Listing 7.11. This automatically created aspect is clean because the contexts of the consistent calls can be grouped, creating only one pointcut and advice for each concern.
Other aspects are not so nice. Aspects that implement a role that has a lot of imposees, tend to be quite large with implementations for the role methods of all the imposees. Aspects that implement consistent behaviour where the consistent call contexts cannot be grouped create a pointcut and advice for each context, which also tends to make them large. All created aspects are listed in Appendix D.

7.4.3 Comparison with AJHOTDRAW

Although the command concerns are not fully refactored because they consist of more than only RSI and CB sort instances, we still try to compare the resulting aspects to the manually created aspects of AJHOTDRAW.

The first thing to note, is that the AJHOTDRAW aspect CommandContracts and the aspect CommandConsistency created by SAIR, are very similar. The main differences are in the pointcut definitions. The AJHOTDRAW pointcuts capture the complete hierar-

Listing 7.11: The aspect of the main command consistency concerns (compacted)
7.4 Discussion

The aspects CommandObserver in SAIR and AJHotDraw are also similar. There are some additional concerns in the AJHotDraw version that represent concerns that were not present in the command concern model. Additionally, for the ‘CommandObserver’ role in the AJHotDraw aspect, non-role methods from AbstractCommand (not in the Command interface) appear in the aspect because they are only used by the role methods. This is in contrast with the SAIR aspect, which contains only the methods from the role interface.

Another observation in the CommandObserver aspect of AJHotDraw is that the (previously virtual) role interface is inside the aspect. The virtual role has indeed been abstracted to a new interface, but this interface is an inner interface in the CommandObserver aspect. A similar situation occurs in the CommandUndo aspect where the virtual role of undoability is also an inner interface of the aspect. The SAIR versions of these abstracted interfaces are the same, only not inside the aspect.

While in AJHotDraw the RSI sort instance of ‘UndoableCommand’ is refactored to a single aspect, the CB sort instances seem to be spread out over several aspects: one for each advised command. At first sight, this is not what we would expect for single sort instances.

However, if we look at the resulting aspect of SAIR for undo concerns, CommandUndoConsistency, we see that this solution is not ideal. The aspect is a very long list of pointcuts and advice that cannot be grouped and are related only be representing the same concern for different commands. It is indeed a nicer solution to create an aspect for each command and put all these aspects in the same package—also a form of modularisation. An additional advantage can be seen in those aspects in AJHotDraw, where the UndoActivity support classes of the commands are also implemented inside ‘their’ aspects, thereby making use of the extra modularity introduced by spreading the aspects.

A final observation is an aspect called CommandAspectsOrdering in AJHotDraw that defines the precedence of the aspects concerning advice. This is important when the calls were migrated from the same context methods, as was the case with many consistent calls in this case study. This is an issue that needs to be addressed manually for now with SAIR, but could be automated based on refactoring order.

7.4.4 Conclusion

The results of the case study performed in this chapter, are summarised in Table 7.1. This table emphasises that most sort instances were successfully refactored to aspects. Only one sort instance could not be refactored because limitations with AspectJ: ‘InitCommand’. The exclusion of ‘Command-MainRole’ was a deliberate choice, not because of limitations.
7.4 Discussion

Table 7.1: Overview of the case study results

<table>
<thead>
<tr>
<th>Sort instance</th>
<th>Results and issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>InitCommand (CB)</td>
<td>Not refactored. AspectJ cannot invoke constructor from advice.</td>
</tr>
<tr>
<td>PreExecutionCheck-ViewNotNull (CB)</td>
<td>Refactored to aspect CommandConsistency. Applied COPY_SUPER_BODY_TO_ADVICE solution to problems with super call.</td>
</tr>
<tr>
<td>Command-MainRole (RSI)</td>
<td>Not refactored. Indicates the main role of commands, not a secondary concern.</td>
</tr>
<tr>
<td>InitUndoSupport (CB)</td>
<td>Refactored to aspect CommandUndoConsistency. Applied MAKE_ASPECT_PRIVILEGED to alleviate visibility problems. Excluded PasteCommand because of nested call.</td>
</tr>
<tr>
<td>SaveFigureStateBefore-Execution (CB)</td>
<td>Refactored to aspect CommandUndoConsistency. Excluded DuplicateCommand and PasteCommand because of nested call. Manually refactored concern-related code to advice for two context methods after refactoring.</td>
</tr>
<tr>
<td>CommandUndoableRole (RSI)</td>
<td>Refactored to aspect CommandUndoRole. Virtual role abstracted to new interface CommandUndo. Manually refactored a role-related field to aspect after refactoring.</td>
</tr>
<tr>
<td>ObservableCommand (RSI)</td>
<td>Refactored to aspect CommandObserver. Virtual role abstracted to new interface ObservableCommand. Applied MAKE_MEMBER_PUBLIC solution to invisible protected methods.</td>
</tr>
<tr>
<td>CommandListener (RSI)</td>
<td>Refactored to aspect CommandObserver. No problems.</td>
</tr>
</tbody>
</table>

Although most sort instances were refactored successfully, the created aspects were of varying quality. Some aspects were very nice and comparable to manually created aspect in AJHOTDRAW. Other aspects were more chaotic, containing large lists of hardly related pointcuts and advice.

The problems with disorderly aspects that where automatically created could possibly be solved by spreading the refactored context elements out over several aspects, like we observed in AJHOTDRAW. If SAIR is to cope with this automatically, some changes are needed, since currently only one target aspect can be handled.

The exclusion of some classes from some of the refactorings (e.g. PasteCommand), could possibly be solved by manual refactoring before and after refactoring with SAIR. Ideally, those manual refactorings should eventually be integrated in the refactoring process, possibly as a solution to a refactoring problem.

All in all, the results are encouraging. The created aspects are a good start to an aspect-oriented application, although the results will still need some attention before working properly (e.g. fixing the precedence of advice and manually refactoring the context methods that...
were excluded). The resulting aspects are a good start to help the developer refactor all the concerns in the application to aspects.
Chapter 8

Conclusions and Future Work

In this chapter, we will conclude this thesis. First, we will evaluate the tool and algorithm based on the case study from the previous chapter. Next, we will discuss the strengths and the weaknesses of the tool and the algorithm, and possible improvements and extensions. Then we will summarise the main contributions of this thesis. After this, we will provide a conclusion with a final answer to the research question. Finally, some suggestions for future work will be presented.

8.1 Evaluation

Section 1.6 explained that the case study would be used to evaluate the algorithm and proof-of-concept tool. Without such an evaluation, it would be hard to determine if the algorithm is successful. The evaluation is based on the two criteria: (1) the success rate of the refactoring and (2) the quality of the created aspects.

8.1.1 Success rate of refactoring

Starting with the success rate of the refactoring, we see that only one of the eight concerns could not be refactored. That is, 87.5% percent of the concerns could be refactored. The reason that one concern could not be refactored, is of a limitation of AspectJ regarding the invocation of constructors from advice, not a limitation of the algorithm or implementation of SAIR.

Of the seven successfully refactored concerns, not all classes could always be refactored. One class, PasteCommand, has been excluded from two refactorings, three other classes, DuplicateCommand, UndoCommand and RedoCommand, have been excluded from one refactoring each. Considering the concerns in question contain around fifteen classes, this is not a bad rate.

Many more problems were detected during the refactoring, as we saw in the previous chapter. However, using the problem resolving capability of SAIR, most of these problems

---

1Eight in total, but not counting the ‘Command-MainRole’ concern, since it was explicitly excluded from refactoring.
could be solved without excluding parts of the refactoring. This has increased the success rate of the refactoring considerably, since otherwise most refactorings would have finished with compile errors.

### 8.1.2 Quality of created aspects

The quality of the created aspects is a more subjective criterion. To quantify the quality of the aspects, we will look at the number of pointcuts and advice (\#PC and \#ADV), the number of inter-type declarations (\#IT), lines of code (LoC) and the overall readability. Table 8.1 lists for each of the four created aspects the values for those quality criteria, except for the subjective criteria of ‘overall readability’.

From this table we can see that two aspects seem very concise: CommandConsistency and CommandUndoRole. Looking at these aspects shows they are indeed very well readable (see Appendix D for the complete listings of all the created aspects).

The quality of CommandObserver is less clear, with 17 inter-type declarations on 84 lines. A visual inspection shows that most of these inter-type declarations are short, readable inter-type method declarations, and that this aspect is also of acceptable quality.

The CommandUndoConsistency aspect however, is not acceptable. Not only does it have a lot of pointcuts and advice, it is also very chaotic. This aspect is of poor quality, and should be refactored as soon as possible. We already saw in the previous chapter that AJHOTDRAW has solved the problem of this chaotic aspect by spreading it out over different aspects: one for each of the commands that it is advising.

Concluding this short evaluation of the tool, we can say that it is indeed capable of successfully refactoring sort instances of RSI and CB sort to aspects. Not all created aspects are of good quality, but a solution has already been identified to fix this problem.

### 8.2 Discussion and Reflection

During the explanation of the algorithms and the implementation of SAIR and during the case study, several observations were made. These observations are reflected upon and discussed in this section.
8.2.1 General observations

Solutions to refactoring problems

When resolving a refactoring problem, not all solutions yield a nice implementation. An example is the `SUPER_REDIRECT_METHOD_IN_ASPECT` solution, that creates an inter-type redirect method that calls the super method. Such solutions should not be considered final solution, but an intermediate step that needs some additional refactoring. Ideally, a better solution should be applied, but better solutions are not always possible. Selecting a less-than-perfect solutions keeps the application from breaking after the refactoring and allows the developer to easily make modifications afterwards than cannot be automated using SAIR.

Other solutions do provide an adequate implementation that needs no manual improvements after refactoring. An example is the `MOVE_SUPER_BODY_TO_ADVICE`, that was successfully used in the case study for the ‘PreExecutionCheck-ViewNotNull’ concern. The resulting code was virtually identical to the manually refactored advice in AJHOTDRAW.

Support for multiple target aspects

When comparing the case study results with AJHOTDRAW we noticed that the `Command-UndoConsistency` aspect created by SAIR corresponded to several aspects in AJHOTDRAW.

Based on this observation, we could improve the aspect implementation created by SAIR if the concern would be spread over multiple aspects. This would require the support for multiple target aspects, or better yet, the automatic creation of target aspects based on the context methods in the sort instance and some package guidance by the developer.

Another solution is to create the concern model in a different way. Instead of creating one concern for all undo-related consistency calls, we could create one CB sort instance for each of the Commands (corresponding to one concern per target aspect), and group them in a composite concern in the concern model. This would result in more refactorings but allows for the creation of spread-out aspects with the current version of SAIR.

Anonymous classes

We have excluded anonymous classes from the refactoring at all times. In the case study however, there were quite a number of commands that were implemented as anonymous classes and they were not considered in the refactoring (see Appendix C for details of the concern model and excluded anonymous classes).

A possible solution for the anonymous classes is transforming them to inner classes and impose a role on or create a pointcut for those inner classes. Fact of the matter is, that they are special cases of commands. It is therefore questionable whether they should be considered in the refactoring at all. Indeed, AJHotDraw seems to have excluded them as well in case of the consistent behaviour of commands.
8.2 Discussion and Reflection

8.2.2 RSI related

Virtual roles

If a virtual role is abstracted to a new interface, this interface is declared parent of the original role, not of all the imposees. The reason behind this was explained in Section 5.3.2. The role methods may still be called on various locations in the program using the original role as type.

A possible solution for this is to determine all those locations where the role methods are called, and insert explicit casts to the new role interface. If we then declare the new interface a parent of all the imposees, this is a valid cast on all occasions. A limitation may be that not all method call sites are known (methods can be called third party code), or are in read-only source files.

Classes as roles

Although in theory, roles in a RSI sort instance can be classes as well as interfaces, SAIR and the algorithm for RSI refactoring only supports interfaces as classes, as discussed in Section 5.3.2. An improvement to the support of RSI sort refactoring would be adding support for classes are roles. This is not as straightforward as it might seem.

With interfaces, all members are public. With classes, role members can also be private, package-visible or protected. This should not create problems with unprivileged aspects, since those members are migrated to the aspect and therefore visible to the aspect.

However, in case the migrated members are referenced by members other than role members, they may be invisible after migration, leading to compile errors. This can be detected as a problem and solved by solutions, but a valid question is whether such role members are really role members, if they are not only referenced by the (migrated) role implementation and also not public.

If the class role is virtual, it can be abstracted to a dedicated interface. Only the signature of the methods should be migrated to the dedicated interface, not the implementation. Abstract role methods can be removed after abstraction to the interface. The original class should become an imposee of the new interface after the abstraction of the virtual role.

If the virtual role members are not public, this can introduce problems since an interface can only contains public members. This means that those role members cannot be abstracted to the interface, or that they should be made public.

An alternative is to abstract the virtual role to an abstract class instead of an interface. This may be a reasonable solution, but if the original class of the virtual role already extends another class, abstracting the role to a dedicated class will introduce multiple inheritance. This is not possible in Java and AspectJ.

Fields as role members

Fields as role-members are currently not supported by SAIR. Role interfaces can have fields and if classes are supported as roles, the possibility for fields as role members becomes more likely. The visibility problems that can occur with those fields have been discussed.
Conclusions and Future Work

8.2 Discussion and Reflection

in the previous section. Implementing fields as role members would require the creation of inter-type field declarations in the aspect.

**Migrate implicit role fields along**

An alternative to supporting fields as role members in classes, is to automatically determine if certain fields are only used by role methods that are being migrated to an aspect. If a certain field is only used by the role method implementations, it will most likely be related to the role implementation of that imposee. An example in the case study is the ‘CommandUndoableRole’ RSI sort, that has a field myUndoableActivity in the AbstractCommand, that is only used by the virtual role. We refactored that field manually to the aspect afterwards.

**Migrate implicit role methods along**

Similarly to automatically migrating role fields along, we can also migrate private methods to the aspect if they are only used inside role methods. Private methods only called by the role methods will probably be part of role implementation of that imposee. For non-private methods, it may be harder to determine if only the role methods call them, and for public methods it is impossible to determine, because not all call sites can be known.

**Put role interface inside aspect**

Comparison with the aspects of AJHOTDRAW revealed that the abstracted virtual role interfaces were implemented as inner interfaces inside the aspect. This creates a closer connection between the role interface and the implementation. This could be easily implemented in SAIR by providing an option to the developer asking where the abstracted interface should be created.

**8.2.3 CB related**

**Consistently invoking superconstructor**

The first concern refactoring we attempted in the case study failed, because of limitations in AspectJ regarding (super)constructor invocation. A possible workaround is the creation and use of an additional method. This additional method contains the body that originally was in the invoked constructor, and is then consistently called instead of the constructor.

Although this will work in most cases, it is not an ideal solution. If the original constructor invocation had parameters (which is likely for an explicit (super)constructor invocation), this call cannot be removed from the context. Java dictates that a constructor must always explicitly call the superconstructor if there is a superconstructor that takes parameters. So although the implementation of the consistent calling can be moved to advice, the (redundant) call to the superconstructor must stay in place.
Preparing the code for refactoring

During the case study we encountered situations in which a consistent call was tangled in the context method in such a way that it could not be migrated to an aspect. Such context methods might benefit from object-oriented refactorings performed before refactoring the concern.

In fact, SAIR already has an implicit mechanism to perform object-oriented refactorings: the problem resolving step. If the refactoring problems can be detected that adequately describe the problem (instead of, for example, the current generic ‘Advising nested call’) and if more complex solutions can be devised, the object-oriented refactorings can be incorporated in the problem step of the sort-based algorithm.

Creating advice for inadvisable contexts

When a ‘Advising nested call’ or ‘Advising neither first nor last call’ problems is determined, the solutions are currently limited to explicitly creating before or after advice. Several other solutions are imaginable that would make it possible to create pointcut and advice for the inadvisable context.

Advice after preceding call  Instead of capturing the context method with a pointcut and create advice for that pointcut, a pointcut can also be created that captures the call immediately preceding the consistent statement. Using after advice for such a pointcut, the consistent statement can be used to advise the context. A requirement is the existence of a method call directly preceding the consistent statement.

Advice before succeeding call  This is similar to the previous solution, only capturing the succeeding call with a pointcut and creating before advice. A requirement is the existence of a method call directly succeeding the consistent statement.

Extract block contents to separate method  When a consistent statement is the first or last in a block inside a method (e.g. an if block), the content of that block can be moved to a different method. Since the consistent statements is now first or last in this new method, it can be advised using before or after advice on this new method.

Move preceding code to along to advice  When a consistent statement is not first in the context method, but the code preceding it is directly related, an option is to migrate the preceding code along with the consistent statement to the advice. This makes it possible to create before advice for a consistent statement that was originally not the first statement in the context method.

This is specifically useful for consistent behaviour that is more than one method call in some cases. An example in the case study was for the sort instance ‘SaveFiguresStateBeforeExecution’, where the consistent statements in CutCommand and DeleteCommand were preceded by related code. This related code was later manually refactored to the created before advice for those context methods.
The drawback with these solutions (except the last), is that they create pointcuts that are different from the expected pointcuts, because they do not capture only the context method. Modifications to the method body may suddenly change the behaviour of the advice, for instance when the preceding call is suddenly removed while it is used in the pointcut and advice.

**Duplicate consistent call**

The CB sort defines a method that is consistently called and the context it is consistently called in. Often this context is a hierarchy—in the JHOTDRAW refactoring it was mainly the Command hierarchy. In SAIR it is assumed that this context consists of methods in which the consistent method is called. In fact, it is assumed that the consistent method is called only once in each context method.

This need not be the case, as we saw in the refactoring of the undo concern of JHOTDRAW. More than one consistent method call in a context method can indicate several things. First, it can be intentional: the consistent method simply needs to be called more than once. Second, it can be an implementation mistake: it should only be called once. Finally, their can be all sorts of structural reasons, e.g. one call in the if block and one in the else block. These situations can be identified as a refactoring problem.

The solution that should be applied depends on the reason of the duplicate call. If the duplication is intentional, one option is to create more than one advice for the method, another option is to put both calls in the same advice if they adjacent in the original context. If the duplication is not intentional, an option is to migrate one of the calls to the advice and leave the others in the context method for future fixing. Finally if the call is duplicated because of structure, an option may be to advice using one call, but remove both calls from the context method.

**Grouping by advice**

The refactoring of the CB sort supports three forms of grouping pointcuts and advice: no grouping, grouping by pointcut and grouping by advice. We used the first two in the case study, but the ‘grouping by advice’ variant was not used. This does not necessarily mean that it is of no use, but if future case studies still make no use of this grouping, it might be not as useful as at first imagined. This may be a reason to remove this unused feature in future.

**Aspect precedence**

In the case study, we refactored several method calls from the same context methods to different aspects. The order of execution of these aspect is nondeterministic by default, possibly leading to undesirable results. The solution is to create an additional aspect in which the order of precedence for the aspects is defined. This additional aspect can be based on the refactoring order. Currently, it is not automated in SAIR, so the developer should keep such an aspect updated during the refactoring of different sorts.
8.2 Discussion and Reflection

8.2.4 Other crosscutting concern sorts

Of the ten remaining sorts besides RSI and CB, four others are implemented in SOQUEt: Redirection Layer, Expose Context (Context Passing), Exception Propagation, and Support Classes for Role Superimposition [40, 39]. It is possible to extend SAIR and implement refactorings for at least three of those four sorts.

Redirection Layer

The Redirection Layer (RL) sort indicates that an interfacing layer forwards calls to dedicated methods of another object. The layer acts as a front-end that accepts calls and redirects them, with or without executing additional functionality. It is typically an indication of Decorator, Adapter or Facade patterns [40].

Refactoring an RL sort instance would create pointcuts for each method that redirects and would migrate the redirection call to around advice. If the redirecting method adds logic, proceed can be called in the advice. Drawback of this approach can be loss of flexibility for Decorator-related instances of this sort [36, 24].

Expose Context

The Expose Context (EC) sort, or Context Passing as it is called in SOQUEt, indicates a chain of callees where the context of the caller is exposed to each of the callees. This is typically done by declaring an additional parameter to pass specific context required to fulfil the (secondary) requirements [40].

Refactoring an EC sort instance would create a pointcut for the caller that has the context and a pointcut for the callee that needs the context. The ‘Wormhole’ pattern can then be applied to pass the context. This pattern is described by Laddad and applied in the example refactoring ‘replace argument trickle by wormhole’ [29, 30].

Exception Propagation

The Exception Propagation (EP) sort indicates the consistent propagation of exceptions in a call chain, when neither the callees and the callers have an appropriate response for the exception. In Java, all non-RuntimeExceptions, called checked exceptions, must either be caught or declared to be thrown by the method using a throws clause. The EP sort therefore indicates the enforced logic of rethrowing checked exceptions when methods are not able to handle them [40].

Refactoring an EP sort instance would make use of the AspectJ declare soft mechanism. This mechanism is used to statically catch unchecked exceptions and rethrow them wrapped in the org.aspectj.lang.SoftException, which is a subclass of RuntimeException. Using this mechanism alleviates the need for rethrowing the exception explicitly, since it is now a runtime exception. A disadvantage is that the identity of the original exception is lost when wrapping it in a SoftException [36].
Support Classes

The Support Classes for Role Superimposition (SC) sort is an alternative to normal RSI that uses nested classes for roles instead of the implementation of interfaces. This explicates the relation between the role and the enclosing class [40].

Unfortunately, instances of the SC sort cannot be easily refactored to an aspect solution. The mechanism that would ideally fit this sort is the introduction of nested classes. AspectJ does not support the introduction of nested classes, making it impossible to straightforwardly implement SC sort instances in an aspect [36].

Although the SC sort cannot be refactored in the ideal way, there are workarounds imaginable. The refactored JHOTDRAW version AJHOTDRAW has implemented these support classes in aspects using normal inner classes, i.e. without introducing them to the original enclosing types. This brings along other problems, since the inner classes of the aspect can no longer access the original surrounding class directly. So although modularity will be improved and refactoring is desirable, it might not be as easily automated as refactorings for the other sorts.

8.2.5 SOQUET

When using SOQUET for the refactorings of SAIR, two issues arose that could be an improvement to SOQUET in general and for its use by SAIR in particular.

Application Programming Interface

One improvement to SOQUET that would benefit SAIR and possible other plug-ins that want to make use of the concern model, is an Application Programming Interface (API). Such an API would allow the other plug-ins to execute the queries and obtain the context elements of sort instances, without using a workaround concern model as we did for SAIR.

The API could also provide functionality for easy modification of the concern model in use. That way, SAIR would be able to create additional sort instance that indicate only the context elements that have been excluded from the refactoring (e.g. when problems occurred) and remove the already refactored instances.

Aspect-aware concern model

Another improvement to SOQUET would be to make it aspect-aware. This way, SAIR would not need to remove the refactored sort instances from the concern model (through the new API) after refactoring. The sort instance would still find the same context elements, even though these elements are now inside aspects.

8.3 Contributions

The contributions of this thesis are threefold. First, a generic algorithm has been developed for the refactoring of sorts. Second, the algorithms for the refactoring of two sorts have
been developed. Third, a working tool has been developed that implements both the generic algorithm as well as the algorithms for the refactoring of the two sorts.

8.3 Contributions

Conclusions and Future Work

8.3.1 Generic algorithm

Chapter 4 describes a generic algorithm for the refactoring of sorts. This algorithm consists of 5 steps: (1) gathering input, (2) determining potential problems, (3) resolving potential problems, (4) applying selected solutions and (5) migrating the sort instance to an aspect. Concrete versions of the algorithm use this generic version and add the details for a specific sort.

A main contribution of this generic algorithm is the refactoring problem framework. This framework allows specific sort refactorings to determine problems and possible solutions before the problems occur. These problems are then resolved by the developer by selecting one of the possible solutions. The selected solutions are then applied before the actual migration, making sure no unexpected problems arise during the migration step.

The possible problems currently implemented, and the conditions in which they can occur were presented in Section 4.3.2. A list of possible solutions for those problems, along with the set of properties of these solutions, was also presented in that section. A complete list of how the problems occur in the refactoring of the two supported sorts and what the possible solutions will do, is given in Appendix B.

8.3.2 Algorithms for RSI and CB sorts

Chapter 5 presents that algorithm for refactoring the RSI sort and Chapter 6 present the algorithms for refactoring the CB sort. These two algorithms show how the input gathering step and the actual migration step depend heavily on the specific sort being refactored.

The determination of problems also depends on the sort being refactored, but it also depends very much on the implementation of the algorithm. Therefore, no explicit algorithms are given for the determination of problems for the RSI and CB sorts. However, Appendix B shows for RSI and CB how each of the problems can occur and which solutions will resolve these problems.

8.3.3 SAIR

The generic algorithm and the algorithms for the RSI and CB sorts have been implemented in the tool SAIR\(^2\). A modest case study on JHOTDRAW was described in Chapter 7. The results of this case study show that the principal ideas behind the tool and the algorithm are indeed successful in migrating crosscutting concerns, expressed as sort instances, to aspects.

The capabilities of the current version of SAIR are summarised in Table 8.2. This table shows the main functional features that can already be used to refactor many RSI and CB sort instances to aspects.

\(^2\)SAIR is available for download on \url{http://swerl.tudelft.nl/bin/view/AMR/WebHome}
Table 8.2: Overview of main capabilities of SAIR

<table>
<thead>
<tr>
<th>Sort</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both</td>
<td>• Detection and solving of visibility problems with fields, methods and classes used in aspect after migration</td>
</tr>
</tbody>
</table>
| RSI  | • Migration of role method implementation to aspect using inter-type declarations  
      • Superimposition of roles on imposees by aspect  
      • Abstraction of virtual role interface to new dedicated interface. |
| CB   | • Migration of consistent call to advice  
      • Grouping pointcuts for context methods with same advice  
      • Grouping advice for pointcuts that can have same advice  
      • Use of super method body in advice instead of call to super method (implemented as problem/solution)  
      • Conversion of local variable to field when used in consistent call (implemented as problem/solution)  
      • Creation of before/after advice explicitly when consistent call nested or not first/last in context method (implemented as problem/solution) |

The limitations of SAIR are summarised in Table 8.3. For each of these limitations, a reason is given. Most of these limitations are also opportunities for improvements in future versions.

8.4 Conclusions

The goal set for this thesis in Section 1.5 in the introduction was providing an answer to the following research question:

How can crosscutting concern sorts be used for aspect-introducing refactoring?

This is not a yes-or-no question with a straightforward answer. The answer actually lies in the algorithms and implementation of SAIR, presented in the Chapters 4, 5 and 6. These chapters show just how crosscutting concern sorts can be used for aspect-introducing refactoring.

We can state here, that these chapters provide indeed a valid answer. The case study in Chapter 7, showed SAIR is capable of refactoring with crosscutting concern sorts as input. This indicates that the algorithm behind SAIR, which is based on sorts, can successfully be used for aspect-introducing refactoring.

Since this is a first attempt at such tool-supported refactoring algorithm for sort-based aspect-introducing refactoring, we can expect that there are limitations. Some of these limitation concern the target aspect language used by SAIR—AspectJ. An example is the impossibility to access super members and invoke constructors from advice. Other limitations are due to SAIR being still a proof-of-concept tool, for example the fact that only interfaces are supported as roles.

Concluding, we can safely say that the current version of SAIR is already a functioning tool to support aspect-introducing refactoring. The problem resolving framework is
### Table 8.3: Overview of current limitations and improvement opportunities

<table>
<thead>
<tr>
<th>Sort</th>
<th>Limitation/opportunity</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both</td>
<td>• Use of anonymous classes in concern not supported.</td>
<td>Anonymous classes cannot be referenced from aspects, making it impossible to create inter-type declarations and hard to created pointcuts. Anonymous classes are special cases.</td>
</tr>
<tr>
<td></td>
<td>• Multiple target aspects, for spreading large concern implementations, not supported</td>
<td>Conceptually, atomic sort instances can be refactored to one aspect each. However, resulting aspects can be large and chaotic. Spreading implementation out over multiple aspects is a solution. Not hard to implement.</td>
</tr>
<tr>
<td>RSI</td>
<td>• Only interfaces as (virtual) roles supported</td>
<td>Misunderstanding of sort definition and resulting simplification of algorithm. Support of classes will make algorithm more complex, not needed for proof-of-concept. Not hard to implement.</td>
</tr>
<tr>
<td></td>
<td>• Only fields supported as (virtual) role members</td>
<td>Misunderstanding of sort definition. Support of fields is mainly interesting when classes are supported as roles, since interfaces usually have less fields. Not hard to implement.</td>
</tr>
<tr>
<td></td>
<td>• Imposing abstracted virtual role on imposes directly not possible, new role is imposed on original role.</td>
<td>Imposing abstracted interface on imposees directly may break method invocations in system. Possible solution is inserting explicit casts for role methods in system. Not hard to implement. Not implemented yet. Can easily be implemented as refactoring option.</td>
</tr>
<tr>
<td></td>
<td>• Abstracted virtual role as inner interface of aspect is not supported.</td>
<td></td>
</tr>
<tr>
<td>CB</td>
<td>• The precedence of the create aspects and advice is not set to any order.</td>
<td>Not implemented yet. Can be solved by creating an additional aspect during refactoring that declares the precedence of created aspects.</td>
</tr>
<tr>
<td></td>
<td>• Superconstructor invocations not supported as consistent call.</td>
<td>Limitation of AspectJ. Possible solution is extraction of superconstructor body to new method, but call may still be required by Java.</td>
</tr>
<tr>
<td></td>
<td>• Consistent calls that are not first or last in context are not easily correctly advised.</td>
<td>Limitation of aspect languages in general. Advice only easily created for first or last method call. Some solutions can be imagined to solve some cases. Hard to implement for all cases.</td>
</tr>
</tbody>
</table>
a powerful and extensible way to handle refactoring problems regarding both the existing object-oriented code as well as generated aspect-oriented code. The resulting aspects are a good step towards a proper aspect-oriented system, although some manual refactoring might still be needed sometimes.

8.5 Future Work

This thesis has shown how to use crosscutting concern sorts as a basis for aspect-introducing refactoring. The proof-of-concept tool SAIR was developed for this purpose. The success of SAIR and the algorithms behind it, lead to the following suggestions for future work. These suggestions are inspired by the limitations and opportunities encountered, discussed in Section 8.2.

8.5.1 Refactoring of other sorts

The successful refactoring of the RSI and CB sorts supported by SAIR provide a promising view on the refactoring of other sorts. We suggest the implementation of refactoring of four sorts as future work. These sorts are Redirection Layer, Expose Context, Exception Propagation and Support Classes. See Section 8.2.4 for a discussion of these sorts and their implementation suggestions.

8.5.2 Improvements to existing algorithms and implementations

In Section 8.2 we noticed several possible improvements and extensions to SAIR and the algorithms. In general, the support for multiple aspect in case of large refactoring would be an improvement.

For the RSI sort refactoring, the improvements include the support of classes as roles and fields as role members, the imposing of abstracted virtual role interface directly on the imposees, the migration of implicit role members along to the aspect and the option to put the abstracted virtual role inside the aspect.

For the CB sort refactoring, the improvements include the automatic creation of a precedence aspect, ways to automatically create better context methods using new problems and solutions, and various solutions that will make inadvisable contexts advisable.

The concrete implementation of these improvements may need some additional research. These improvements and the accompanying research are suggested as future work.

8.5.3 Improvements to SOQUET

Since SAIR is heavily dependent on SOQUET for providing the concern model and therefore the sort input, improvements to SOQUET would also benefit SAIR. The improvements discussed in Section 8.2.5 include an API for executing queries and an API for modifying the concern model. Aspect-awareness may be an improvement that is useful to SOQUET even without SAIR, since it would enable the documentation feature of SOQUET for AspectJ programs.
Bibliography


[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Reading, Massachusetts, 1994.


Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

**AJDT** AspectJ Development Tools Plug-in for Eclipse that makes Eclipse an integrated development environment for AspectJ.

**AO(P)** Aspect-oriented (programming). Aspect-oriented programming is a paradigm in which CCCs can be modularised using modules called *aspects*.

**API** Application Programming Interface A source code interface that operating systems, libraries or other programs provide to support requests by computer programs for services to be made.

**Aspect-introducing refactoring** Refactoring method that extracts CCCs from OO context to aspects, making the application aspect-oriented. See Section 2.3.2.

**AspectJ** Aspect-oriented programming language that extends Java with aspects. AspectJ is becoming the *de facto* standard of AOP, and is the most widely used implementation.

**AST** Abstract Syntax Tree. An abstract tree representation of a source file. The tree is finite, with each node representing a programming structure in the source file. The root node represents the complete source file.

**CB** Consistent Behavior. Sort indicating the consistent calling of the same method from different locations. One of the six sorts supported by

**CCC** Crosscutting concern. This indicates a concern in the code that crosscuts the primary functionality of the modules it resides in.

**EC** Expose Context. One of the six sorts supported by SOQUET, where it is called *Context Passing*.

**Eclipse** Eclipse [10] is an open-source software framework written primarily in Java. The capabilities of Eclipse can be extended by installing plug-ins written for Eclipse, such as development toolkits for various programming languages. The most popular distribution of Eclipse is that of software development kit (SDK) for Java.
EP Exception Propagation. One of the six sorts supported by SOQUET.

GoF Gang-of-Four. Refers to the four authors of the original design pattern book: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides[19].

IDE Integrated Development Environment. An IDE is an application that provides various bundled facilities to computer programmers for software development. These facilities commonly include a source file editor, a compiler and/or interpreter, build automation tools and a debugger.

Java Java is an object-oriented programming language originally developed by Sun Microsystems and released in 1995 as a core component of Sun’s Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode which can run on any Java virtual machine (JVM) regardless of computer architecture[58].


LTK Language Toolkit. Plug-in for Eclipse that provides a language independent tooling layer. The current version allows for the development of refactorings that look and feel like native Eclipse refactorings.

OO(P) Object-oriented (programming). Object-oriented programming is a paradigm in which functionality is modularised using . . .

PDE Plug-in Development Environment Plug-in for Eclipse that facilitates the creation of other plug-ins.

RCP Rich Client Platform. A piece of software that functions as a basis for the development of other applications. The RCP provides common functionality, e.g. a portable widget toolkit (GUI) and file handling systems. Programs built with an RCP are often portable to many operating systems, as is the case with programs built on the Eclipse framework.

RL Redirection Layer. One of the six sorts supported by SOQUET.

RSI Role Superimposition. Sort indicating a secondary role that is imposed on classes. One of the six sorts supported by SOQUET. Implemented as a refactoring in SAIR SOQUET. Implemented as a refactoring in SAIR

SC Support Classes. Short for Support Classes for Role Superimposition. One of the six sorts supported by SOQUET.

SDK Software Development Kit an SDK is a set of development tools that allows a software engineer to create applications for a certain software package, software framework,
Glossary

hardware platform, computer system, video game console, operating system, or similar platform [59]. For example, Eclipse provides an SDK for developing programs in Java.

Sort Crosscutting concern sort. A crosscutting concern sort is a generic description of a class of concerns. These concerns are atomic and share three properties: (1) an intent, (2) a implementation idiom in OO languages and (3) a desired AOP mechanism that modularises sort instances. See Section 2.4.

Sort instance . A concrete occurrence in the code of a crosscutting concern sort.

SoQueT Sorts Query Tool. SoQueT is a tool for consistently describing and documenting CCCs as sort instances [51]. See Section 2.4.
Appendix B

Solutions per Problem

This appendix describes the refactoring problems that can occur for the RSI sort and the CB sort, along with a short example and the set of solutions. This set of problems and solutions is one of the contributions of this thesis.

The problems are described per sort in the following way. First, a short description of when the problem will occur is given. Next, a simple example of the occurrence of the problem is presented and explained. Finally, the solutions that will resolve the problem are listed, with a short explanation of the consequences of applying the solution. If needed, this explanation refers back to the example.

Future developments will extend the catalogue of refactoring problem in this appendix with new problems and solutions for the two currently supported sorts, and new problems and solution for the refactoring of other sorts as those refactorings are implemented in SAIR.

B.1 Role Superimposition

This section contains the set of refactoring problems that can occur when refactoring an RSI sort instance. The role interface that is used in all the examples is shown in Listing B.1.

The problems in this section are based on the implementation of these problems in SAIR. Since SAIR can only handle interfaces as roles, problems that would occur with classes as roles are not considered. See also Section 5.3.2.

```java
package example.role;
public interface IRole{
    // only one role method
    void roleMethod();
}
```

Listing B.1: Role interface used in examples for RSI problems
B.1 Role Superimposition

Listing B.2: Example of imposee using private method

```java
package example.role;
public class Imposee implements IRole {
    private void someMethod(){
        // ... [body] ...
    }
    public void roleMethod() {
        // ... [body] ...
    someMethod();
    }
}
```

Listing B.3: Example of inter-type method using invisible method

```java
package example.aspect;
import example.role.Imposee;
aspect RSIExampleAspect{
    public void Imposee.roleMethod() {
        // ... [body] ...
    someMethod(); // <= problem: invisible
    }
}
```

B.1.1 Access of private method

The ‘access of private method’ problem occurs in the RSI refactoring when the implementation of a role method contains a call to a private method. Moving this implementation to an inter-type method declaration will then result in the usage of a private method from the aspect, which is a problem if the aspect is not privileged.

Example of ‘access of private method’

Suppose one of the imposees is implemented as shown in Listing B.2. The role method contains a call to a private method. If this method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.3. This shows how an unprivileged aspect will try to access a private method, leading to a compile error.

Solutions for ‘access of private method’

The solution set of the ‘access of private method’ problem contains the following five solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the private method to public, making the method accessible from the aspect without problems. In the example, the visibility of `someMethod()` will be changed to `public`.  

124
## Solutions per Problem

### B.1 Role Superimposition

```java
package example.role;
public class Imposee implements IRole {
    private int someField = 0;

    public void roleMethod() {
        // ... [body] ...
        someField = 1;
    }
}
```

Listing B.4: Example of imposee using private field

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the private method without problems. In the example, RSIExampleAspect will be made privileged.

**EXCLUDE_METHOD** Excludes the method that has the call to the private method from the refactoring. In the example, Imposee.roleMethod() will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the private call from the refactoring. In the example, Imposee will be excluded from the refactoring.

### B.1.2 Access of private field

The ‘access of private field’ problem occurs in the RSI refactoring when the implementation of a role method contains a reference to or assignment of a private field. Moving this implementation to an inter-type method declaration will then result in the usage of a private field from the aspect, which is a problem if the aspect is not privileged.

**Example of ‘access of private field’**

Suppose one of the imposees is implemented as shown in Listing B.4. The role method contains a reference to a private field. If this method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.5. This shows how an unprivileged aspect will try to access a private field, leading to a compile error.

**Solutions for ‘access of private field’**

The solution set of the ‘access of private field’ problem contains the following six solutions:

- **IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

- **MAKE_MEMBER_PUBLIC** Changes the visibility of the private field to public, making the field accessible from the aspect without problems. In the example, the visibility of someField will be changed to public.
B.1 Role Superimposition

Solutions per Problem

Listing B.5: Example of inter-type method using invisible field

```java
package example.aspect;
import example.role.Imposee;

aspect RSIExampleAspect{
  public void Imposee.roleMethod() {
    // ...
    
    someField = 1; // <= problem: invisible
  }
}
```

CREATE GETTER_SETTER FOR FIELD Creates public getter and setter methods for the private field and substitutes direct access to the field with the getter/setter. In the example, a setter would be created for `someField` and would be used to replace the direct assignment.

MAKE ASPECT PRIVILEGED Makes the target aspect privileged, allowing the aspect to access the private field without problems. In the example, `RSIExampleAspect` will be made privileged.

EXCLUDE METHOD Excludes the method that has the reference to the private field from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

EXCLUDE TYPE Excludes the class that contains the method that contains the reference to the private field from the refactoring. In the example, `Imposee` will be excluded from the refactoring.

B.1.3 Access of package-visible method

The ‘access of package-visible method’ problem occurs in the RSI refactoring when the implementation of a role method contains a call to a package-visible method. Moving this implementation to an inter-type method declaration will then result in the usage of a package-visible method from the aspect, which is a problem if the aspect is in a different package and is not privileged.

Example of ‘access of package-visible method’

Suppose one of the imposees is implemented as shown in Listing B.6. The role method contains a call to a package-visible method. If this role method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.3, but `someMethod()` has package visibility. This shows how an unprivileged aspect will try to call a package-visible method in another package, leading to a compile error.
Solutions for ‘access of package-visible method’

The solution set of the ‘access of package-visible method’ problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible method to public, making the method accessible from the aspect without problems. In the example, the visibility of `someMethod()` will be changed to `public`.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible method without problems. In the example, the aspect `RSIExampleAspect` will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible method, making the method accessible from the aspect without problems. In the example, the `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.

**EXCLUDE_METHOD** Excludes the method that has the call to the package-visible method from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the package-visible call from the refactoring. In the example, `Imposee` will be excluded from the refactoring.

### B.1.4 Access of package-visible field

The ‘access of package-visible field’ problem occurs in the RSI refactoring when the implementation of a role method contains a reference to or assignment of a package-visible field. Moving this implementation to an inter-type method declaration will then result in the usage of a package-visible field from the aspect, which is a problem if the aspect is in a different package and is not privileged.
Example of ‘access of package-visible field’

Suppose one of the imposees is implemented as shown in Listing B.7. The role method contains a reference to a package-visible field. If this method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.5, but someField has package visibility. This shows how an unprivileged aspect will try to access a package-visible field in another package, leading to a compile error.

### Solutions for ‘access of package-visible field’

The solution set of the ‘access of package-visible field’ problem contains the following seven solutions:

- **IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.
- **MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to `public`.
- **CREATE_GETTER_SETTER_FOR_FIELD** Creates public getter and setter methods for the package-visible field and substitutes direct access to the field with the getter/setter. In the example, a setter would be created for `someField` and would be used to replace the direct assignment.
- **MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible field without problems. In the example, `RSIExampleAspect` will be made privileged.
- **MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible field, making the field accessible from the aspect without problems. In the example, the `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.
- **EXCLUDE_METHOD** Excludes the method that has the reference to the package-visible field from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

---

Listing B.7: Example of imposee using package-visible field

```java
package example.role;

public class Imposee implements IRole {
    int someField = 0;

    public void roleMethod() {
        // ... [body] ...
        someField = 1;
    }
}
```

Example of ‘access of package-visible field’

The solution set of the ‘access of package-visible field’ problem contains the following seven solutions:

- **IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.
- **MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to `public`.
- **CREATE_GETTER_SETTER_FOR_FIELD** Creates public getter and setter methods for the package-visible field and substitutes direct access to the field with the getter/setter. In the example, a setter would be created for `someField` and would be used to replace the direct assignment.
- **MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible field without problems. In the example, `RSIExampleAspect` will be made privileged.
- **MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible field, making the field accessible from the aspect without problems. In the example, the `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.
- **EXCLUDE_METHOD** Excludes the method that has the reference to the package-visible field from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

---

Example of ‘access of package-visible field’

Suppose one of the imposees is implemented as shown in Listing B.7. The role method contains a reference to a package-visible field. If this method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.5, but someField has package visibility. This shows how an unprivileged aspect will try to access a package-visible field in another package, leading to a compile error.

### Solutions for ‘access of package-visible field’

The solution set of the ‘access of package-visible field’ problem contains the following seven solutions:

- **IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.
- **MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to `public`.
- **CREATE_GETTER_SETTER_FOR_FIELD** Creates public getter and setter methods for the package-visible field and substitutes direct access to the field with the getter/setter. In the example, a setter would be created for `someField` and would be used to replace the direct assignment.
- **MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible field without problems. In the example, `RSIExampleAspect` will be made privileged.
- **MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible field, making the field accessible from the aspect without problems. In the example, the `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.
- **EXCLUDE_METHOD** Excludes the method that has the reference to the package-visible field from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

---

Listing B.7: Example of imposee using package-visible field

```java
package example.role;

public class Imposee implements IRole {
    int someField = 0;

    public void roleMethod() {
        // ... [body] ...
        someField = 1;
    }
}
```
Listing B.8: Example of imposee using protected method

```java
package example.role;
public class Imposee implements IRole {
    protected void someMethod() {
        // ... (body) ...
    }
    public void roleMethod() {
        // ... (body) ...
        someMethod();
    }
}
```

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the package-visible field from the refactoring. In the example, Imposee will be excluded from the refactoring.

### B.1.5 Access of protected method

The ‘access of protected method’ problem occurs in the RSI refactoring when the implementation of a role method contains a call to a protected method. Moving this implementation to an inter-type method declaration will then result in the usage of a package-visible method from the aspect, which is a problem if the aspect is in a different package and is not privileged.

**Example of ‘access of protected method’**

Suppose one of the imposees is implemented as shown in Listing B.8. The role method contains a call to a package-visible method. If this role method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.3, but `someMethod()` has protected visibility. This shows how an unprivileged aspect will try to call a protected method in another package, leading to a compile error.

**Solutions for ‘access of protected method’**

The solution set of the ‘access of protected method’ problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected method to public, making the method accessible from the aspect without problems. In the example, the visibility of `someMethod()` will be changed to `public`.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected method without problems. In the example, RSIExample-Aspect will be made privileged.
### B.1 Role Superimposition

#### Solutions per Problem

---

```java
package example.role;

public class Imposee implements IRole {
    protected int someField = 0;

    public void roleMethod() {
        // ... {body} ...
        someField = 1;
    }
}
```

Listing B.9: Example of imposee using protected field

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the protected method, making the method accessible from the aspect without problems. In the example, the `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.

**EXCLUDE_METHOD** Excludes the method that has the call to the protected method from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the protected call from the refactoring. In the example, `Imposee` will be excluded from the refactoring.

#### B.1.6 Access of protected field

The ‘access of protected field’ problem occurs in the RSI refactoring when the implementation of a role method contains a reference to or assignment of a protected field. Moving this implementation to an inter-type method declaration will then result in the usage of a protected field from the aspect, which is a problem if the aspect is in a different package and is not privileged.

**Example of ‘access of protected field’**

Suppose one of the imposees is implemented as shown in Listing B.9. The role method contains a reference to a protected field. If this method is migrated to an inter-type method as part of the refactoring, the implementation will look like Listing B.5, but `someField` has protected visibility. This shows how an unprivileged aspect will try to access a protected field in another package, leading to a compile error.

**Solutions for ‘access of protected field’**

The solution set of the ‘access of protected field’ problem contains the following seven solutions:
**Solutions per Problem**

**B.1 Role Superimposition**

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to public.

**CREATE_GETTER_SETTER_FOR_FIELD** Creates public getter and setter methods for the protected field and substitutes direct access to the field with the getter/setter. In the example, a setter would be created for `someField` and would be used to replace the direct assignment.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected field without problems. In the example, `RSIExampleAspect` will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the protected field, making the field accessible from the aspect without problems. In the example, the `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.

**EXCLUDE_METHOD** Excludes the method that has the reference to the protected field from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the protected field from the refactoring. In the example, `Imposee` will be excluded from the refactoring.

**B.1.7 Access of a private inner class**

The ‘access of a private inner class’ problem occurs in the RSI refactoring in two situations. One is when the implementation of a role method contains a reference to a private inner class (*Case I*). Moving this implementation to an inter-type method declaration will then result in the usage of a private inner class from the aspect, which is a problem if the aspect is not privileged. The other situation is when one of the imposees is a private inner class (*Case II*). Creating a declare parents and inter-type methods for this imposee will result in the usage of a private inner class from the aspect, which is a problem if the aspect is not privileged.

**Example of ‘access of private inner class’ Case I**

As an example of the first case where a private inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.10. The role method contains a reference to a private inner class. If this method is migrated to an aspect as part of the refactoring, the implementation will look like Listing B.11. This shows how an unprivileged aspect will try to use a private inner class, leading to a compile error. Note that the reference
### Solutions per Problem

#### B.1 Role Superimposition

**Listed B.10**: Example of imposee *using* private inner class (Case I)

```java
package example.role;
public class Imposee implements IRole {
    private class Inner {
        // ... [body] ...
    }

    public void roleMethod() {
        // ... [body] ...
        Inner ic = new Inner();
    }
}
```

**Listed B.11**: Example of inter-type method using invisible inner class (Case I)

```java
package example.aspect;
import example.role.Imposee;
aspect RSIExampleAspect{
    public void Imposee.roleMethod() {
        // ... [body] ...
        Imposee.Inner ic = new Imposee.Inner(); // <= problem: invisible
    }
}
```

to the inner class has already been qualified using the surrounding class name `Imposee`, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

**Solutions for ‘access of private inner class’ Case I**

The solution set of the ‘access of private inner class’ Case I problem contains the following five solutions:

- **IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

- **MAKE_MEMBER_PUBLIC** Changes the visibility of the private inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of `Inner` will be changed to `public`.

- **MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the private inner class without problems. In the example, `RSIExampleAspect` will be made privileged.

- **EXCLUDE_METHOD** Excludes the method that has the reference to the private inner class from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.
Solutions per Problem

B.1 Role Superimposition

Listing B.12: Example of imposee being private inner class (Case II)

```java
package example.role;
public class SomeClass {
    private class Imposee implements IRole {
        public void roleMethod() {
            // ... [body] ...
        }
    }
}
```

Listing B.13: Example of declare parents and inter-type method on invisible inner class (Case II)

```java
package example.aspect;
import example.role.SomeClass;
aspect RSIEExampleAspect{
    declare parents: SomeClass.Imposee implements IRole;
    // ^^^^^^ <= problem: invisible
    public void SomeClass.Imposee.roleMethod() {
        // ^^^^^^ <= problem: invisible
        // ... [body] ...
    }
}
```

**EXCLUDE**

EXcludes the class that contains the method that contains the reference to the private inner class from the refactoring. In the example, Imposee will be excluded from the refactoring.

**Example of ‘access of private inner class’ Case II**

As an example of the second case where a private inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.12. The imposee is a private inner class. If this imposee is migrated to an aspect as part of the refactoring, the implementation will look like Listing B.13. This shows how an unprivileged aspect will try to declare parents and inter-type methods on a private inner class, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name Imposee, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

**Solutions for ‘access of private inner class’ Case II**

The solution set of the ‘access of private inner class’ Case II problem contains the following four solutions:

**IGNORE**

Ignores the access problem. This will lead to a compile error in the aspect after refactoring.
MAKE_MEMBER_PUBLIC Changes the visibility of the private inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of SomeClass.Imposee will be changed to public.

MAKE_ASPECT_PRIVILEGED Makes the target aspect privileged, allowing the aspect to access the private inner class without problems. In the example, RSIExample-Aspect will be made privileged.

EXCLUDE_TYPE Excludes the imposee that is a private inner class from the refactoring. In the example, SomeClass.Imposee will be excluded from the refactoring.

B.1.8 Access of a package-visible inner class

The ‘access of a package-visible inner class’ problem occurs in the RSI refactoring in two situations. One is when the implementation of a role method contains a reference to a package-visible inner class (Case I). Moving this implementation to an inter-type method declaration will then result in the usage of a package-visible inner class from the aspect, which is a problem if the aspect is not privileged and in another package. The other situation is when one of the imposees is a package-visible inner class (Case II). Creating a declare parents and inter-type methods for this imposee will result in the usage of a package-visible inner class from the aspect, which is a problem if the aspect is not privileged and in another package.

Example of ‘access of package-visible inner class’ Case I

As an example of the first case where a package-visible inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.14. The role method contains a reference to a package-visible inner class (Case I). Moving this implementation to an inter-type method declaration will then result in the usage of a package-visible inner class from the aspect, which is a problem if the aspect is not privileged and in another package. The other situation is when one of the imposees is a package-visible inner class (Case II). Creating a declare parents and inter-type methods for this imposee will result in the usage of a package-visible inner class from the aspect, which is a problem if the aspect is not privileged and in another package.

Solutions for ‘access of package-visible inner class’ Case I

The solution set of the ‘access of package-visible inner class’ Case I problem contains the following six solutions:

IGNORE Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

MAKE_MEMBER_PUBLIC Changes the visibility of the package-visible inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of Inner will be changed to public.
Solutions per Problem

B.1 Role Superimposition

```java
package example.role;
public class Imposee implements IRole {
    class Inner {
        // ... [body] ...
    }

    public void roleMethod() {
        // ... [body] ...
        Inner ic = new Inner();
    }
}
```

Listing B.14: Example of imposee using package-visible inner class (Case I)

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible inner class without problems. In the example, the aspect RSIExampleAspect will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible inner class, making the inner class accessible from the aspect without problems. In the example, the RSIExampleAspect will be moved from the example.aspects package to the example.role package.

**EXCLUDE_METHOD** Excludes the method that has the reference to the package-visible inner class from the refactoring. In the example, Imposee.roleMethod() will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the package-visible inner class from the refactoring. In the example, Imposee will be excluded from the refactoring.

**Example of ‘access of package-visible inner class’ Case II**

As an example of the second case where a package-visible inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.15. The imposee is a package-visible inner class. If this imposee is migrated to an aspect as part of the refactoring, the implementation will look like Listing B.13, but the inner class SomeClass.Imposee has package visibility. This shows how an unprivileged aspect will try to declare parents and inter-type methods on a package-visible inner class in another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name Imposee, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

**Solutions for ‘access of package-visible inner class’ Case II**

The solution set of the ‘access of package-visible inner class’ Case II problem contains the following five solutions:
B.1 Role Superimposition

Solutions per Problem

B.1.9 Access of a protected inner class

The ‘access of a protected inner class’ problem occurs in the RSI refactoring in two situations. One is when the implementation of a role method contains a reference to a protected inner class (Case I). Moving this implementation to an inter-type method declaration will then result in the usage of a protected inner class from the aspect, which is a problem if the aspect is not privileged and not in the same package. The other situation is when one of the imposees is a protected inner class (Case II). Creating a declare parents and inter-type methods for this imposee will result in the usage of a protected inner class from the aspect, which is a problem if the aspect is not privileged and in another package.

Example of ‘access of protected inner class’ Case I

As an example of the first case where a protected inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.16. The role method contains a reference to a protected inner class. If this method is migrated to an aspect as part of

```java
package example.role;
public class SomeClass {
    class Imposee implements IRole {
        public void roleMethod() {
            // ... [body] ...
        }
    }
}
```

Listing B.15: Example of imposee being package-visible inner class (Case II)

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of `SomeClass.Imposee` will be changed to public.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible inner class without problems. In the example, the aspect `RSIExampleAspect` will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible inner class, making the inner class accessible from the aspect without problems. In the example, the aspect `RSIExampleAspect` will be moved from the `example.aspects` package to the `example.role` package.

**EXCLUDE_TYPE** Excludes the imposee that is a package-visible inner class from the refactoring. In the example, `SomeClass.Imposee` will be excluded from the refactoring.
the refactoring, the implementation will look like Listing B.11, but Inner has protected visibility. This shows how an unprivileged aspect will try to use a protected inner class from another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name Imposee, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of protected inner class’ Case I

The solution set of the ‘access of protected inner class’ Case I problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of Inner will be changed to public.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected inner class without problems. In the example, RSIExampleAspect will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the protected inner class, making the inner class accessible from the aspect without problems. In the example, the RSIExampleAspect will be moved from the example.aspects package to the example.role package.

**EXCLUDE_METHOD** Excludes the method that has the reference to the protected inner class from the refactoring. In the example, Imposee.roleMethod() will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the protected inner class from the refactoring. In the example, Imposee will be excluded from the refactoring.
Example of ‘access of protected inner class’ Case II

As an example of the second case where a protected inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.17. The imposee is a protected inner class. If this imposee is migrated to an aspect as part of the refactoring, the implementation will look like Listing B.13, but SomeClass.Imposee has protected visibility. This shows how an unprivileged aspect will try to declare parents and inter-type methods on a protected inner class in another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name Imposee, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of protected inner class’ Case II

The solution set of the ‘access of protected inner class’ Case II problem contains the following five solutions:

- **IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

- **MAKE_MEMBER_PUBLIC** Changes the visibility of the protected inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of SomeClass.Imposee will be changed to public.

- **MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected inner class without problems. In the example, RSIExampleAspect will be made privileged.

- **MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the protected inner class, making the inner class accessible from the aspect without problems. In the example, the RSIExampleAspect will be moved from the example.aspects package to the example.role package.

- **EXCLUDE_TYPE** Excludes the imposee that is a protected inner class from the refactoring. In the example, SomeClass.Imposee will be excluded from the refactoring.
B.10 Access of a package-visible class

The ‘access of a package-visible class’ problem occurs in the RSI refactoring in two situations. One is when the implementation of a role method contains a reference to a package-visible class (Case I). Moving this implementation to an inter-type method declaration will then result in the usage of a package-visible class from the aspect, which is a problem if the aspect is not in the same package. The other situation is when one of the imposees is a package-visible class (Case II). Creating a declare parents and inter-type methods for this imposee will result in the usage of a package-visible class from the aspect, which is a problem if the aspect is not in the same package.

Note that it doesn’t matter whether the aspect is privileged or not. Privileged aspects can access non-public members of other classes, not non-public classes themselves if these are not inner classes.

Example of ‘access of package-visible class’ Case I

As an example of the first case where a package-visible inner class poses a problem, suppose one of the imposees is implemented as shown in Listing B.18. The role method contains a reference to a package-visible class. If this method is migrated to an aspect as part of the refactoring, the implementation will look like Listing B.19. This shows how an aspect will try to use a package-visible class from another package, leading to a compile error.

Solutions for ‘access of package-visible class’ Case I

The solution set of the ‘access of private inner class’ Case I problem contains the following five solutions:

IGNORE Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

MAKE_MEMBER_PUBLIC Changes the visibility of the private inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of Inner will be changed to public.
B.1 Role Superimposition

Solutions per Problem

Listing B.19: Example of inter-type method using package-visible class (Case I)

```java
package example.aspect;
import example.role.Imposee;
import example.role.SomeClass; // <= problem: invisible
aspect RSIExampleAspect{
  public void Imposee.roleMethod() {
    // ... [body] ...
    SomeClass sc = new SomeClass(); // <= problem: invisible
  }
}
```

Listing B.20: Example of imposee being package-visible class (Case II)

```java
package example.role;
class Imposee implements IRole {
  public void roleMethod() {
    // ... [body] ...
  }
}
```

MOVE_ASPECT_TO_PACKAGE Moves the target aspect to the same package as the accessed package-visible class, making the class accessible from the aspect without problems. In the example, the aspect RSIExampleAspect will be moved from the example.aspects package to the example.role package.

EXCLUDE_METHOD Excludes the method that has the reference to the private inner class from the refactoring. In the example, Imposee.roleMethod() will be excluded from the refactoring.

EXCLUDE_TYPE Excludes the class that contains the method that contains the reference to the private inner class from the refactoring. In the example, Imposee will be excluded from the refactoring.

Example of ‘access of package-visible class’ Case II

As an example of the second case where a package-visible class poses a problem, suppose one of the imposees is implemented as shown in Listing B.20. The imposee is a package-visible class. If this imposee is migrated to an aspect as part of the refactoring, the implementation will look like Listing B.21. This shows how an aspect will try to declare parents and inter-type methods on a package-visible class from another package, leading to a compile error.

Solutions for ‘access of package-visible class’ Case II

The solution set of the ‘access of package-visible inner class’ Case II problem contains the following four solutions:
Solutions per Problem  

B.1 Role Superimposition

```java
package example.aspect;
import example.role.Imposee;
aspect RSIExampleAspect{
    declare parents: Imposee implements IRole;
    // ^^^^^^ <= problem: invisible
    public void Imposee.roleMethod() {
        // ^^^^^^ <= problem: invisible
        // ... [body] ...
    }
}
```

Listing B.21: Example of declare parents and inter-type method on package-visible class (Case II)

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible class to public, making the class accessible from the aspect without problems. In the example, the visibility of Imposee will be changed to public.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the accessed package-visible class, making the class accessible from the aspect without problems. In the example, the aspect RSIExampleAspect will be moved from the example.aspects package to the example.role package.

**EXCLUDE_TYPE** Excludes the imposee that is a package-visible class from the refactoring. In the example, Imposee will be excluded from the refactoring.
B.2 Consistent Behavior

This section contains the set of refactoring problems that can occur when refactoring a CB sort instance. Listing B.22 shows the example class containing an example context method that is used in most of the examples. The consistently called method is a public method taking one int as a parameter. The target aspect used in the examples with the pointcut matching the example context method is shown in Listing B.23.

B.2.1 Access of private method

The ‘access of private method’ problem occurs in the CB refactoring when the consistent statement in a context method contains a call to a private method. Moving this consistent statement to advice will then result in the usage of a private method from the aspect, which is a problem if the aspect is not privileged.

Example of ‘access of private method’

Suppose the consistent statement is implemented as shown in Listing B.24. The consistent statement contains a call to a private method. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.25. This shows how an unprivileged aspect will try to access a private method, leading to a compile error.
Solutions per Problem  
B.2 Consistent Behavior

```java
package example.cb;
public class ExampleClass {
    private int someMethod() {
        // ... [body] ...
    }
    public void contextMethod() {
        // ... [body] ...
        consistentlyCalled(someMethod()); // <= consistent statement
    }
}
```

Listing B.24: Example of consistent statement containing a call to a private method

```java
package example.aspect;
import example.cb.ExampleClass;
aspect CBExampleAspect{
    // ... [pointcut 'example'] ...

    after(ExampleClass ec) : example(ec) {
        ec.consistentlyCalled(ec.someMethod());
        // ******* <= problem: invisible
    }
}
```

Listing B.25: Example of advice using an invisible method

Solutions for ‘access of private method’

The solution set of the ‘access of private method’ problem contains the following five solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the private method to public, making the method accessible from the aspect without problems. In the example, the visibility of `someMethod()` will be changed to `public`.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the private method without problems. In the example, `CBExampleAspect` will be made privileged.

**EXCLUDE_METHOD** Excludes the method that has the call to the private method from the refactoring. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the private call from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.
B.2 Consistent Behavior Solutions per Problem

```java
package example.cb;
public class ExampleClass {
    private int someField = 0;

    public void contextMethod() {
        // ... [body] ...
        consistentlyCalled(someField); // <= consistent statement
    }
}
```

Listing B.26: Example of consistent statement referencing a private field

```java
package example.aspect;
import example.cb.ExampleClass;
aspect CBExampleAspect{
    // ... [pointcut 'example'] ...

    after(ExampleClass ec) : example(ec){
        ec.consistentlyCalled(ec.somefield);
        // ----------- <= problem: invisible
    }
}
```

Listing B.27: Example of advice using an invisible field

B.2.2 Access of private field

The ‘access of private field’ problem occurs in the CB refactoring when the consistent statement in a context method contains a reference to a private field. Moving this consistent statement to advice will then result in the usage of a private field from the aspect, which is a problem if the aspect is not privileged.

Example of ‘access of private field’

Suppose the consistent statement is implemented as shown in Listing B.26. The consistent statement contains a reference to a private field. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.27. This shows how an unprivileged aspect will try to access a private field, leading to a compile error.

Solutions for ‘access of private field’

The solution set of the ‘access of private field’ problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the private field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to public.
B.2 Consistent Behavior

The ‘access of package-visible method’ problem occurs in the CB refactoring when the consistent statement in a context method contains a call to a package-visible method. Moving this consistent statement to advice will then result in the usage of a package-visible method from the aspect, which is a problem if the aspect is not privileged and not in the same package.

Example of ‘access of package-visible method’

Suppose the consistent statement is implemented as shown in Listing B.28. The consistent statement contains a call to a package-visible method. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.25, but someMethod() has package visibility. This shows how an unprivileged aspect will try to access a package-visible method, leading to a compile error.
Solutions for ‘access of package-visible method’

The solution set of the ‘access of package-visible method’ problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible method to public, making the method accessible from the aspect without problems. In the example, the visibility of `someMethod()` will be changed to `public`.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible method without problems. In the example, the aspect `CBExampleAspect` will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible method, making the method accessible from the aspect without problems. In the example, the `CBExampleAspect` will be moved from the `example.aspects` package to the `example.cb` package.

**EXCLUDE_METHOD** Excludes the method that has the call to the package-visible method from the refactoring. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the package-visible call from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.

### B.2.4 Access of package-visible field

The ‘access of package-visible field’ problem occurs in the CB refactoring when the consistent statement in a context method contains a reference to a package-visible field. Moving this consistent statement to advice will then result in the usage of a package-visible field from the aspect, which is a problem if the aspect is not privileged and not in the same package.

**Example of ‘access of package-visible field’**

Suppose the consistent statement is implemented as shown in Listing B.29. The consistent statement contains a reference to a package-visible field. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.27, but the field `someField` will have package visibility. This shows how an unprivileged aspect will try to access a package-visible field from another package, leading to a compile error.
Solutions for ‘access of package-visible field’

The solution set of the ‘access of package-visible field’ problem contains the following seven solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to public.

**CREATE_GETTER_SETTER_FOR_FIELD** Creates public getter and setter methods for the package-visible field and substitutes direct access to the field with the getter/setter. In the example, a getter would be created for `someField` and the reference in the consistent statement would be replaced by a call to this getter.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible field without problems. In the example, `CBExampleAspect` will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible field, making the field accessible from the aspect without problems. In the example, the `CBExampleAspect` will be moved from the `example.aspects` package to the `example.cb` package.

**EXCLUDE_METHOD** Excludes the method that has the reference to the package-visible field from the refactoring. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the package-visible field from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.
B.2 Consistent Behavior

B.2.5 Access of protected method

The ‘access of protected method’ problem occurs in the CB refactoring when the consistent statement in a context method contains a call to a protected method. Moving this consistent statement to advice will then result in the usage of a protected method from the aspect, which is a problem if the aspect is not privileged and not in the same package.

Example of ‘access of protected method’

Suppose the consistent statement is implemented as shown in Listing B.30. The consistent statement contains a call to a protected method. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.25, but the method someMethod() will have protected visibility. This shows how an unprivileged aspect will try to access a protected method in another package, leading to a compile error.

Solutions for ‘access of protected method’

The solution set of the ‘access of protected method’ problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected method to public, making the method accessible from the aspect without problems. In the example, the visibility of someMethod() will be changed to public.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected method without problems. In the example, RSIEExampleAspect will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the type containing the package-visible method, making the method accessible from the aspect without problems. In the example, the CBExampleAspect will be moved from the example.aspects package to the example.cb package.
**Solutions per Problem**

**B.2 Consistent Behavior**

```java
package example.cb;
public class ExampleClass {
    protected int someField = 0;

    public void contextMethod() {
        // ... [body] ...
        consistentlyCalled(someField); // <= consistent statement
    }
}
```

Listing B.31: Example of consistent statement referencing a protected field

**EXCLUDE_METHOD** Excludes the method that has the call to the protected method from the refactoring. In the example, `Imposee.roleMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the protected call from the refactoring. In the example, `Imposee` will be excluded from the refactoring.

### B.2.6 Access of protected field

The ‘access of protected field’ problem occurs in the CB refactoring when the consistent statement in a context method contains a reference to a protected field. Moving this consistent statement to advice will then result in the usage of a protected field from the aspect, which is a problem if the aspect is not privileged and not in the same package.

**Example of ‘access of protected field’**

Suppose the consistent statement is implemented as shown in Listing B.31. The consistent statement contains a reference to a protected field. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.27, but the field `someField` will have protected visibility. This shows how an unprivileged aspect will try to access a protected field in another package, leading to a compile error.

**Solutions for ‘access of protected field’**

The solution set of the ‘access of protected field’ problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected field to public, making the field accessible from the aspect without problems. In the example, the visibility of `someField` will be changed to public.
B.2 Consistent Behavior  Solutions per Problem

CREATE_GETTER_SETTER_FOR_FIELD Creates public getter and setter methods for the
protected field and substitutes direct access to the field with the getter/setter. In the
example, a getter would be created for someField and the reference in the consist-
tent statement would be replaced by a call to this getter.

MAKE_ASPECT_PRIVILEGED Makes the target aspect privileged, allowing the aspect to
access the protected field without problems. In the example, CBExampleAspect
will be made privileged.

MOVE_ASPECT_TO_PACKAGE Moves the target aspect to the same package as the type
containing the protected field, making the field accessible from the aspect with-
out problems. In the example, the CBExampleAspect will be moved from the
example.aspects package to the example.cb package.

EXCLUDE_METHOD Excludes the method that has the reference to the protected field from
the refactoring. In the example, ExampleClass.contextMethod will be ex-
cluded from the refactoring.

EXCLUDE_TYPE Excludes the class that contains the method that contains the reference
to the protected field from the refactoring. In the example, ExampleClass will be
excluded from the refactoring.

B.2.7 Access of a private inner class

The ‘access of a private inner class’ problem occurs in the CB refactoring in two situations.
One is when the consistent statement contains a reference to a private inner class (Case I).
Moving this implementation to advice will then result in the usage of a private inner class
from the aspect, which is a problem if the aspect is not privileged. The other situation is
when the context method is the method of a private inner class (Case II). Creating pointcut
and advice for the consistent statement in this context method will result in the usage of a
private inner class from the aspect, which is a problem if the aspect is not privileged.

Example of ‘access of private inner class’ Case I

As an example of the first case where a private inner class poses a problem, suppose the
consistent statement is implemented as shown in Listing B.32. The consistent statement
contains a reference to a private inner class. If this method is migrated to advice as part
of the refactoring, the implementation will look like Listing B.33. This shows how an
unprivileged aspect will try to access a private inner class, leading to a compile error. Note
that the reference to the inner class has already been qualified using the surrounding class
name ExampleClass, otherwise the problem wouldn’t be that the inner class is invisible,
but that it cannot be resolved at all.

Solutions for ‘access of private inner class’ Case I

The solution set of the ‘access of private inner class’ Case I problem contains the following
five solutions:
Solutions per Problem  

B.2 Consistent Behavior

```
package example.cb;
public class ExampleClass {
    private class Inner{
        // ... [body] ...
        public static int FIELD = 0;
    }
    public void contextMethod() {
        // ... [body] ...
        consistentlyCalled(Inner.FIELD); // <= consistent statement
    }
}
```

Listing B.32: Example of consistent statement using a private inner class

```
package example.aspect;
import example.cb.ExampleClass;
aspect CBExampleAspect{
    // ... [pointcut 'example'] ...
    after(ExampleClass ec) : example(ec){
        ec.consistentlyCalled(ExampleClass.Inner.FIELD);
        // ~~~~ <= problem: invisible
    }
}
```

Listing B.33: Example of advice using an invisible inner class

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the private inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of Inner will be changed to public.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the private inner class without problems. In the example, CBExampleAspect will be made privileged.

**EXCLUDE_METHOD** Excludes the method that has the reference to the private inner class from the refactoring. In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the private inner class from the refactoring. In the example, ExampleClass will be excluded from the refactoring.
Example of ‘access of private inner class’ Case II

As an example of the second case where a private inner class poses a problem, suppose the context method is implemented as shown in Listing B.34. The context method is a method of a private inner class. If pointcut and advice are created for this context method as part of the refactoring, the implementation will look like Listing B.35. This shows how an unprivileged aspect will try to access a private inner class, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name `ExampleClass`, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of private inner class’ Case II

The solution set of the ‘access of private inner class’ Case II problem contains the following four solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.
MAKE_MEMBER_PUBLIC Changes the visibility of the private inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of SomeClass.ExampleClass will be changed to public.

MAKE_ASPECT_PRIVILEGED Makes the target aspect privileged, allowing the aspect to access the private inner class without problems. In the example, CBExampleAspect will be made privileged.

EXCLUDE_TYPE Excludes the private inner class that contains the context method from the refactoring. In the example, SomeClass.ExampleClass will be excluded from the refactoring.

B.2.8 Access of a package-visible inner class

The ‘access of a package-visible inner class’ problem occurs in the CB refactoring in two situations. One is when the consistent statement contains a reference to a package-visible inner class (Case I). Moving this implementation to advice will then result in the usage of a package-visible inner class from the aspect, which is a problem if the aspect is not privileged and in another package. The other situation is when the context method is the method of a package-visible inner class (Case II). Creating pointcut and advice for the consistent statement in this context method will result in the usage of a package-visible inner class from the aspect, which is a problem if the aspect is not privileged and in another package.

Example of ‘access of package-visible inner class’ Case I

As an example of the first case where a package-visible inner class poses a problem, suppose the consistent statement is implemented as shown in Listing B.36. The consistent statement contains a reference to a package-visible inner class. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.33, but the class Inner is package-visible. This shows how an unprivileged aspect will try to access a package-visible inner class in another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name ExampleClass, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of package-visible inner class’ Case I

The solution set of the ‘access of package-visible inner class’ Case I problem contains the following six solutions:

IGNORE Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

MAKE_MEMBER_PUBLIC Changes the visibility of the package-visible inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of Inner will be changed to public.
B.2 Consistent Behavior Solutions per Problem

1 package example.cb;
2 public class ExampleClass {
3     class Inner {
4         // ... [body] ...
5         public static int FIELD = 0;
6     }
7     public void contextMethod() {
8         // ...[body] ...
9         consistentlyCalled(Inner.FIELD); // <= consistent statement
10     }
11 }

Listing B.36: Example of consistent statement using a package-visible inner class

MAKE_ASPECT_PRIVILEGED Makes the target aspect privileged, allowing the aspect to access the package-visible inner class without problems. In the example, CBExampleAspect will be made privileged.

MOVE_ASPECT_TO_PACKAGE Moves the target aspect to the same package as the class containing the package-visible inner class, making the inner class accessible from the aspect without problems. In the example, the CBExampleAspect will be moved from the example.aspects package to the example.cb package.

EXCLUDE_METHOD Excludes the method that has the reference to the package-visible inner class from the refactoring. In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

EXCLUDE_TYPE Excludes the class that contains the method that contains the reference to the package-visible inner class from the refactoring. In the example, ExampleClass will be excluded from the refactoring.

Example of ‘access of package-visible inner class’ Case II

As an example of the second case where a package-visible inner class poses a problem, suppose the context method is implemented as shown in Listing B.37. The context method is a method of a package-visible inner class. If pointcut and advice are created for this context method as part of the refactoring, the implementation will look like Listing B.35, but the class Inner is package-visible. This shows how an unprivileged aspect will try to access a package-visible inner class from another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name ExampleClass, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of package-visible inner class’ Case II

The solution set of the ‘access of package-visible inner class’ Case II problem contains the following five solutions:
B.2 Consistent Behavior

Listing B.37: Example of consistent statement inside a package-visible inner class

```java
package example.cb;
public class SomeClass {
    class ExampleClass {
        public void contextMethod() {
            // ... [body] ...
            consistentlyCalled(0); // <= consistent statement
        }
    }
}
```

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of `SomeClass.ExampleClass` will be changed to `public`.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the package-visible inner class without problems. In the example, `CBExampleAspect` will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the class containing the package-visible inner class, making the inner class accessible from the aspect without problems. In the example, the `CBExampleAspect` will be moved from the `example.aspects` package to the `example.cb` package.

**EXCLUDE_TYPE** Excludes the package-visible inner class that contains the context method from the refactoring. In the example, `SomeClass.ExampleClass` will be excluded from the refactoring.

B.2.9 Access of a protected inner class

The `access of a protected inner class` problem occurs in the CB refactoring in two situations. One is when the consistent statement contains a reference to a protected inner class (`Case I`). Moving this implementation to advice will then result in the usage of a protected inner class from the aspect, which is a problem if the aspect is not privileged and in another package. The other situation is when the context method is the method of a protected inner class (`Case II`). Creating pointcut and advice for the consistent statement in this context method will result in the usage of a protected inner class from the aspect, which is a problem if the aspect is not privileged and in another package.

**Example of ‘access of protected inner class’ Case I**

As an example of the first case where a protected inner class poses a problem, suppose the consistent statement is implemented as shown in Listing B.38. The consistent statement
B.2 Consistent Behavior Solutions per Problem

contains a reference to a protected inner class. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.33, but the class Inner is protected. This shows how an unprivileged aspect will try to access a protected inner class in another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name ExampleClass, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of protected inner class’ Case I

The solution set of the ‘access of protected inner class’ Case I problem contains the following six solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of Inner will be changed to public.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected inner class without problems. In the example, CBExampleAspect will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the class containing the protected inner class, making the inner class accessible from the aspect without problems. In the example, the CBExampleAspect will be moved from the example.aspects package to the example.cb package.

**EXCLUDE_METHOD** Excludes the method that has the reference to the protected inner class from the refactoring. In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the protected inner class from the refactoring. In the example, ExampleClass will be excluded from the refactoring.
Example of ‘access of protected inner class’ Case II

As an example of the second case where a protected inner class poses a problem, suppose the context method is implemented as shown in Listing B.39. The context method is a method of a protected inner class. If pointcut and advice are created for this context method as part of the refactoring, the implementation will look like Listing B.35, but the class Inner is protected. This shows how an unprivileged aspect will try to access a protected inner class from another package, leading to a compile error. Note that the reference to the inner class has already been qualified using the surrounding class name ExampleClass, otherwise the problem wouldn’t be that the inner class is invisible, but that it cannot be resolved at all.

Solutions for ‘access of protected inner class’ Case II

The solution set of the ‘access of protected inner class’ Case II problem contains the following five solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the protected inner class to public, making the class accessible from the aspect without problems. In the example, the visibility of SomeClass.ExampleClass will be changed to public.

**MAKE_ASPECT_PRIVILEGED** Makes the target aspect privileged, allowing the aspect to access the protected inner class without problems. In the example, CBExampleAspect will be made privileged.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the class containing the protected inner class, making the inner class accessible from the aspect without problems. In the example, the CBExampleAspect will be moved from the example.aspects package to the example.cb package.

**EXCLUDE_TYPE** Excludes the protected inner class that contains the context method from the refactoring. In the example, SomeClass.ExampleClass will be excluded from the refactoring.

Listing B.39: Example of consistent statement inside a protected inner class
B.2 Consistent Behavior

### Solutions per Problem

#### B.2.10 Access of a package-visible class

The ‘access of a package-visible class’ problem occurs in the CB refactoring in two situations. One is when the consistent statement contains a reference to a package-visible class (Case I). Moving this implementation to advice will then result in the usage of a package-visible class from the aspect, which is a problem if the aspect is not in the same package. The other situation is when the context method is the method of a package-visible class (Case II). Creating pointcut and advice for the consistent statement in this context method will result in the usage of a package-visible class from the aspect, which is a problem if the aspect is not in the same package.

##### Example of ‘access of package-visible class’ Case I

As an example of the first case where a package-visible class poses a problem, suppose the consistent statement is implemented as shown in Listing B.40. The consistent statement contains a reference to a package-visible class. If this method is migrated to advice as part of the refactoring, the implementation will look like Listing B.41. This shows how an aspect will try to access a package-visible class in another package, leading to a compile error.

```java
package example.cb;
public class ExampleClass {
   public void contextMethod() {
      // ... [body] ...
      consistentlyCalled(SomeClass.FIELD); // <= consistent statement
   }
}
package example.cb
class SomeClass {
   // ... [body] ...
   public static int FIELD = 0;
}
```

**Listing B.40:** Example of consistent statement using a package-visible class

```java
package example.aspect;
import example.cb.ExampleClass;
import example.cb.SomeClass; // <= problem: invisible
aspect CBExampleAspect{
   // ... [pointcut 'example'] ...
   after(ExampleClass ec) : example(ec){
      ec.consistentlyCalled(SomeClass.FIELD);
      // ------------ <= problem: invisible
   }
}
```

**Listing B.41:** Example of advice using a package-visible class
Solutions per Problem  B.2 Consistent Behavior

```java
package example.cb;

class ExampleClass {
    public void contextMethod() {
        // ... \(\text{body}\) ...
        consistentlyCalled(0); // <= consistent statement
    }
}
```

Listing B.42: Example of consistent statement inside a package-visible class

Solutions for ‘access of package-visible inner class’ Case I

The solution set of the ‘access of package-visible class’ Case I problem contains the following five solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible class to public, making the class accessible from the aspect without problems. In the example, the visibility of `SomeClass` will be changed to `public`.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the package-visible class, making the class visible from the aspect without problems. In the example, the `CBExampleAspect` will be moved from the `example.aspects` package to the `example.cb` package.

**EXCLUDE_METHOD** Excludes the method that has the reference to the package-visible class from the refactoring. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the method that contains the reference to the package-visible class from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.

Example of ‘access of package-visible class’ Case II

As an example of the second case where a package-visible class poses a problem, suppose the context method is implemented as shown in Listing B.42. The context method is a method of a package-visible class. If pointcut and advice are created for this context method as part of the refactoring, the implementation will look like Listing B.43, but the class `Inner` is package-visible. This shows how an aspect will try to access a package-visible class from another package, leading to a compile error.

Solutions for ‘access of package-visible class’ Case II

The solution set of the ‘access of package-visible class’ Case II problem contains the following four solutions:
B.2 Consistent Behavior Solutions per Problem

### B.2.11 Access of super field

The ‘Access of super field’ problem occurs in the CB refactoring when the consistent statement in a context contains an explicit reference to a super field. Moving this statement to advice will result in the usage of a super field from the advice, which is not possible in AspectJ, and therefore a problem.

#### Example of ‘Access of super field’

Suppose the consistent statement is implemented as shown in Listing B.44. The consistent statement contains an explicit reference to the super field field (which is not the same as this.field). If this consistent call is migrated to advice as part of the refactoring, the implementation will look like Listing B.45. This shows how an aspect will try to access a super field directly from advice, leading to a compile error.

```java
package example.aspect;
import example.cb.ExampleClass; // <= problem: invisible
aspect CBExampleAspect{
    pointcut example(ExampleClass ec):
        // -------- <= problem: invisible
        target(ec) && within(ec) && execution(public void contextMethod());

    after(ExampleClass ec) : example(ec){
        // -------- <= problem: invisible
        ec.consistentlyCalled(0);
    }
}
```

Listing B.43: Example of pointcut and advice for a package-visible class

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_MEMBER_PUBLIC** Changes the visibility of the package-visible class to public, making the class accessible from the aspect without problems. In the example, the visibility of SomeClass.ExampleClass will be changed to public.

**MOVE_ASPECT_TO_PACKAGE** Moves the target aspect to the same package as the package-visible class, making the class accessible from the aspect without problems. In the example, the CBExampleAspect will be moved from the example.aspects package to the example.cb package.

**EXCLUDE_TYPE** Excludes the package-visible class that contains the context method from the refactoring. In the example, SomeClass.ExampleClass will be excluded from the refactoring.
Solutions for ‘Access of super field’

The solution set of the ‘access of super field’ problem contains the following four solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**CREATE_GETTER_SETTER_FOR_FIELD** Creates public getter and setter methods for the super field in the same class as the context method. In the example, a getter and setter for `super.field` would be created in the `ExampleClass` class the direct access of the super field in the consistent statement would be replaced by a call to this getter.

**EXCLUDE_METHOD** Excludes the method that has the consistent call containing the reference to the super field from the refactoring. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the context method that has the reference to the super field in the consistent statement from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.

---

Listing B.44: Example of consistent statement referencing super field explicitly

```java
package example.cb;
public class ExampleClass extends SuperClass {
    public int field = 1;
    public void contextMethod() {
        // ... [body] ...
        consistentlyCalled(super.field); // <= consistent statement
    }
}
package example.cb
public class SuperClass {
    public int field = 0;
    // ... [body] ...
}
```

Listing B.45: Example of advice containing reference to super field

```java
package example.aspect;
import example.cb.ExampleClass;
aspect CBExampleAspect{
    // ... [pointcut 'example'] ...
    after(ExampleClass ec) : example(ec){
        ec.con sistently Called (super.field);
        // ...... <= problem: not possible
    }
}
```
B.2 Consistent Behavior

B.2.12 Access of super method

The ‘Access of super method’ problem occurs in the CB refactoring when the consistent statement in a context contains an explicit call to a super method. Moving this statement to advice will result in the usage of a super method from the advice, which is not possible in AspectJ, and therefore a problem.

Example of ‘Access of super method’

Suppose the consistent statement is implemented as shown in Listing B.46. The consistent statement contains an explicit reference to the super method `getInt()` (which is not the same as `this.getInt()`). If this consistent call is migrated to advice as part of the refactoring, the implementation will look like Listing B.47. This shows how an aspect will try to access a super method directly from advice, leading to a compile error.
Solutions for ‘Access of super method’

The solution set of the ‘access of super method’ problem contains the following five solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**COPY_SUPER_BODY_TO_ADVICE** Copies the body of the super method that is called to the advice, instead of the super call. This is only applicable if the consistent statement is nothing more than a super call. In the example, this is not the case. If the consistent statement would be `super.someMethod()`, this solution would copy the body of `someMethod()` to the advice, instead of the statement `super.someMethod()`. Note that new problems may occur in the body that is copied (e.g. visibility problems) so the problems are refreshed.

**SUPER_REDIRECT_METHOD_IN_ASPECT** Creates an inter-type redirect method in the aspect that calls the super method. The original call to the super method in the consistent statement is then replaced by a call to this redirect method. This solution works, because it is possible to call super methods from inter-type methods, but not from advice.

**EXCLUDE_METHOD** Excludes the method that has the consistent call containing the call of the super method from the refactoring. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the context method that has the call to the super method in the consistent statement from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.

B.2.13 Advising nested call

The ‘Advising nested call’ problem occurs in the CB refactoring when the the consistent statement is not in the direct body of the context method (but e.g. inside an if-block). No advice can be automatically created for such a nested consistent statement.

**Example of ‘Advising nested call’**

Suppose the consistent statement is implemented as shown in Listing B.48. The consistent statement is nested inside an if block. It is not possible to create advice for this consistent statement, since neither before nor after advice are adequate, although after seems more reasonable since the call is actually last in the method, though nested.

**Solutions for ‘Advising nested call’**

The solution set of the ‘advising neither first nor last call’ problem contains the following five solutions:
B.2 Consistent Behavior

B.2.14 Advising neither first nor last call

The ‘Advising neither first nor last call’ problem occurs in the CB refactoring when the consistent statement is not the first nor the last statement in the body of the context method. No advice can be automatically created for such a statement.

Example of ‘Advising neither first nor last call’

Suppose the consistent statement is implemented as shown in Listing B.49. The consistent statement is not first or last in the context method body. It is not possible to create advice for this consistent statement, since neither before nor after advice are adequate, although after seems more reasonable since the call is the last-but-one call in the method.

Listing B.48: Example of a nested consistent statement

```java
package example.cb;

class ExampleClass {
    public void contextMethod() {
        // ...
        if (condition) {
            // ...
            consistentlyCalled(0); // <= consistent statement
        }
    }
}
```

---

**IGNORE** Ignores the access problem. This will actually exclude the context method from the refactoring, because no advice can be created at all. In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

**ADVISE_AS_BEFORE** Creates before advice for the consistent statement, as if it was the first call in the context method. In the example, this is not really a suitable option.

**ADVISE_AS_AFTER** Creates after advice for the consistent statement, as if it was the last call in the context method. In the example, this is an option if the condition of the if block doesn’t really have an influence on the consistent call.

**EXCLUDE_METHOD** Excludes the context method that contains the nested consistent call.

In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the context method that contains the nested consistent statement from the refactoring. In the example, ExampleClass will be excluded from the refactoring.
Solutions for ‘Advising neither first nor last call’

The solution set of the ‘advising neither first nor last call’ problem contains the following five solutions:

**IGNORE** Ignores the access problem. This will actually exclude the context method from the refactoring, because no advice can be created at all. In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

**ADVISE AS BEFORE** Creates before advice for the consistent statement, as if it was the first call in the context method. In the example, this is not really a suitable option.

**ADVISE AS AFTER** Creates after advice for the consistent statement, as if it was the last call in the context method. In the example, this is an option if the method call anotherMethod() is not dependent on the consistent call and the two calls can switch place.

**EXCLUDE METHOD** Excludes the context method that contains the nested consistent call. In the example, ExampleClass.contextMethod() will be excluded from the refactoring.

**EXCLUDE TYPE** Excludes the class that contains the context method that contains the nested consistent statement from the refactoring. In the example, ExampleClass will be excluded from the refactoring.

B.2.15 Access of local variable

The ‘Access of local variable’ problem occurs in the CB refactoring when the consistent statement uses a variable local to the context method. Moving the statement to advice will result in the usage of a local variable from the advice, which is a problem since the local variable is not visible in the advice.

**Example of ‘Access of local variable’**

Suppose the consistent statement is implemented as shown in Listing B.50. The consistent call uses the local variable local. If this consistent call is migrated to advice as part of the refactoring, the implementation will look like Listing B.51. This shows how an aspect will
try to access variable that is local to the context method, leading to a compile error. This is because no pointcut can be created that exposes the local variable to the advice.

**Solutions for ‘Access of local variable’**

The solution set of the ‘access of local variable’ problem contains the following four solutions:

**IGNORE** Ignores the access problem. This will lead to a compile error in the aspect after refactoring.

**MAKE_LOCAL_FIELD** Creates a private field in the class surrounding the context method and replaces the references to the local variable with references to that field. Since fields can be used from within advice, this solves the local variable problem. Note that a private field may introduce visibility problems, which is why the set of problems is refreshed. In the example, the local variable `local` will be replace by a field.

**EXCLUDE_METHOD** Excludes the context method that contains the consistent call that uses the local variable. In the example, `ExampleClass.contextMethod()` will be excluded from the refactoring.

**EXCLUDE_TYPE** Excludes the class that contains the context method that contains the consistent statement that uses a local variable from the refactoring. In the example, `ExampleClass` will be excluded from the refactoring.
Appendix C

JHOTDRAW Concern Model

This appendix lists the details of the SoQUET concern model used in the case study. The concern model contains the Role Superposition (RSI) and Consistent Behavior (CB) sort instances of the ‘Command’ concern in JHOTDRAW 6.0b1.

If any of the imposees (RSI) or context methods (CB) appears in the original version of the concern model, but not in the SAIR compatible version (see Section 4.4.4), this is indicated and explained.

Commands concern model

Command-MainRole – Role Superimposition sort instance

Sort context: Project ‘JHotDraw 6.0b1’
Role interface: org.jhotdraw.util.Command
Virtual role members:

- execute()
- isExecutable()
- getDrawingEditor()
- name()

Imposees:

- org.jhotdraw.standard.AbstractCommand
- org.jhotdraw.standard.AlignCommand
- org.jhotdraw.standard.BringToFrontCommand
- org.jhotdraw.standard.ChangeAttributeCommand
- org.jhotdraw.standard.FigureTransferCommand
- org.jhotdraw.standard.CopyCommand
- org.jhotdraw.standard.CutCommand
• org.jhotdraw.standard.DeleteCommand
• org.jhotdraw.standard.DuplicateCommand
• org.jhotdraw.standard.PasteCommand
• org.jhotdraw.standard.SelectAllCommand
• org.jhotdraw.standard.SendToBackCommand
• org.jhotdraw.standard.ToggleGridCommand
• org.jhotdraw.figures.GroupCommand
• org.jhotdraw.figures.InsertImageCommand
• org.jhotdraw.figures.UngroupCommand
• org.jhotdraw.util.RedoCommand
• org.jhotdraw.util.UndoCommand
• org.jhotdraw.util.UndoableCommand
• org.jhotdraw.contrib.zoom.ZoomCommand
• Various anonymous classes (not included in the SAIR version of the concern model, since anonymous classes are excluded anyway (see Section 4.4.4)).

InitCommand – Consistent Behavior sort instance


AbstractCommand(String, DrawingEditor)

Context methods:

• org.jhotdraw.standard.AlignCommand.
  AlignCommand(Alignment, DrawingEditor)
• org.jhotdraw.standard.BringToFrontCommand.
  BringToFrontCommand(String, DrawingEditor)
• org.jhotdraw.standard.ChangeAttributeCommand.
  ChangeAttributeCommand(String, FigureAttributeConstant, Object, DrawingEditor)
• org.jhotdraw.standard.FigureTransferCommand.
  FigureTransferCommand(String, DrawingEditor)
• org.jhotdraw.standard.SelectAllCommand.
  SelectAllCommand(String, DrawingEditor)
• org.jhotdraw.standard.SendToBackCommand.
  SendToBackCommand(String, DrawingEditor)
• org.jhotdraw.standard.ToggleGridCommand.
  ToggleGridCommand(String, DrawingEditor, Point)
• org.jhotdraw.figures.GroupCommand.
  GroupCommand(String, DrawingEditor)
• org.jhotdraw.figures.InsertImageCommand.
  InsertImageCommand(String, DrawingEditor)
• org.jhotdraw.figures.UngroupCommand.
  UngroupCommand(String, DrawingEditor)
• org.jhotdraw.util.RedoCommand.
  RedoCommand(String, DrawingEditor)
• org.jhotdraw.util.UndoCommand.
  UndoCommand(String, DrawingEditor)

PreExecutionCheck-ViewNotNull – Consistent behavior sort instance
  
  **Sort context:** Hierarchy of org.jhotdraw.util.Command.
  
  **Consistent method:** org.jhotdraw.util.AbstractCommand.
  execute()
  
  **Context methods:**
  
  • org.jhotdraw.standard.AlignCommand.execute()
  • org.jhotdraw.standard.BringToFrontCommand.execute()
  • org.jhotdraw.standard.ChangeAttributeCommand.
    execute()
  • org.jhotdraw.standard.CopyCommand.execute()
  • org.jhotdraw.standard.CutCommand.execute()
  • org.jhotdraw.standard.DeleteCommand.execute()
  • org.jhotdraw.standard.DuplicateCommand.execute()
  • org.jhotdraw.standard.PasteCommand.execute()
  • org.jhotdraw.standard.SelectAllCommand.execute()
  • org.jhotdraw.standard.SendToBackCommand.execute()
  • org.jhotdraw.standard.ToggleGridCommand.execute()
  • org.jhotdraw.figures.GroupCommand.execute()
  • org.jhotdraw.figures.InsertImageCommand.execute()
  • org.jhotdraw.figures.UngroupCommand.execute()
  • org.jhotdraw.util.RedoCommand.execute()
  • org.jhotdraw.util.UndoCommand.execute()
  • org.jhotdraw.util.UndoableCommand.execute() (not included
    in the SAIR version of the concern model.\(^1\))

\(^1\)This seems to be a bug in SoQUET. Even though the method AbstractCommand.execute() is not
called in this context method, only the method Command.execute(), this method is still found by SoQUET
as a context method. Because this is incorrect, this context method is not in the SAIR compatible version of the
concern model.
• org.jhotdraw.contrib.zoom.ZoomCommand.execute()

PostExecutionNotification-UpdateToCmdChanges – Consistent Behavior sort instance


Consistent method: org.jhotdraw.framework.DrawingView.checkDamage()

Context methods:
• org.jhotdraw.standard.AlignCommand.execute()
• org.jhotdraw.standard.BringToFrontCommand.execute()
• org.jhotdraw.standard.ChangeAttributeCommand.execute()
• org.jhotdraw.standard.CutCommand.execute()
• org.jhotdraw.standard.DeleteCommand.execute()
• org.jhotdraw.standard.DuplicateCommand.execute()
• org.jhotdraw.standard.PasteCommand.execute()
• org.jhotdraw.standard.SelectAllCommand.execute()
• org.jhotdraw.standard.SendToBackCommand.execute()
• org.jhotdraw.figures.GroupCommand.execute()
• org.jhotdraw.figures.InsertImageCommand.execute()
• org.jhotdraw.figures.UngroupCommand.execute()
• org.jhotdraw.util.RedoCommand.execute()
• org.jhotdraw.util.UndoCommand.execute()

UndoSupport

CommandUndoableRole Role Superimposition sort instance

Sort context: Hierarchy of org.jhotdraw.util.Command

Role interface: org.jhotdraw.util.Command

Virtual role members:
• getUndoActivity()
• setUndoActivity(Undoable)

Imposees:
• org.jhotdraw.standard.AbstractCommand
• org.jhotdraw.standard.AlignCommand
• org.jhotdraw.standard.BringToFrontCommand
JHotDraw Concern Model

- `org.jhotdraw.standard.ChangeAttributeCommand`
- `org.jhotdraw.standard.FigureTransferCommand`
- `org.jhotdraw.standard.CopyCommand`
- `org.jhotdraw.standard.CutCommand`
- `org.jhotdraw.standard.DeleteCommand`
- `org.jhotdraw.standard.DuplicateCommand`
- `org.jhotdraw.standard.PasteCommand`
- `org.jhotdraw.standard.SelectAllCommand`
- `org.jhotdraw.standard.SendToBackCommand`
- `org.jhotdraw.standard.ToggleGridCommand`
- `org.jhotdraw.figures.GroupCommand`
- `org.jhotdraw.figures.InsertImageCommand`
- `org.jhotdraw.figures.UngroupCommand`
- `org.jhotdraw.util.RedoCommand`
- `org.jhotdraw.util.UndoCommand`
- `org.jhotdraw.util.UndoableCommand`
- `org.jhotdraw.contrib.zoom.ZoomCommand`
- Various anonymous classes (not included in the SAIR version of the concern model, since anonymous classes are excluded anyway (see Section 4.4.4)).

InitUndoSupport  Consistent Behavior sort instance

**Sort context:** Hierarchy of `org.jhotdraw.util.Command`

**Consistent method:** `org.jhotdraw.util.Command.setUndoActivity(Uno`doable)`

**Context methods:**

- `org.jhotdraw.standard.AlignCommand.execute()`
- `org.jhotdraw.standard.BringToFrontCommand.execute()`
- `org.jhotdraw.standard.ChangeAttributeCommand.execute()`
- `org.jhotdraw.standard.CutCommand.execute()`
- `org.jhotdraw.standard.DeleteCommand.execute()`
- `org.jhotdraw.standard.DuplicateCommand.execute()`
- `org.jhotdraw.standard.PasteCommand.execute()`
- `org.jhotdraw.standard.SelectAllCommand.execute()`
- `org.jhotdraw.standard.SendToBackCommand.execute()`
• org.jhotdraw.figures.GroupCommand.execute()
• org.jhotdraw.figures.InsertImageCommand.execute()
• org.jhotdraw.figures.UngroupCommand.execute()

**SaveFigureStateBeforeExecution  Consistent Behavior sort instance**

**Sort context:** Hierarchy of org.jhotdraw.util.Command.

**Consistent method:** org.jhotdraw.util.Undoable.
setAffectedFigures(FigureEnumeration)

**Context methods:**
• org.jhotdraw.standard.AlignCommand.execute()
• org.jhotdraw.standard.BringToFrontCommand.execute()
• org.jhotdraw.standard.ChangeAttributeCommand.execute()
• org.jhotdraw.standard.CutCommand.execute()
• org.jhotdraw.standard.DeleteCommand.execute()
• org.jhotdraw.standard.DuplicateCommand.execute()
• org.jhotdraw.standard.PasteCommand.execute()
• org.jhotdraw.standard.SelectAllCommand.execute()
• org.jhotdraw.standard.SendToBackCommand.execute()
• org.jhotdraw.figures.GroupCommand.execute()
• org.jhotdraw.figures.UngroupCommand.execute()

**CommandObserver**

**ObservableCommand  Role Superimposition sort instance**

**Sort context:** Hierarchy of org.jhotdraw.util.Command

**Role interface:** org.jhotdraw.util.Command

**Virtual role members:**
• addCommandListener(CommandListener)
• removeCommandListener()

**Imposees:**
• org.jhotdraw.standard.AbstractCommand
• org.jhotdraw.standard.AlignCommand
• org.jhotdraw.standard.BringToFrontCommand
• org.jhotdraw.standard.ChangeAttributeCommand
JHotDraw Concern Model

- org.jhotdraw.standard.FigureTransferCommand
- org.jhotdraw.standard.CopyCommand
- org.jhotdraw.standard.CutCommand
- org.jhotdraw.standard.DeleteCommand
- org.jhotdraw.standard.DuplicateCommand
- org.jhotdraw.standard.PasteCommand
- org.jhotdraw.standard.SelectAllCommand
- org.jhotdraw.standard.SendToBackCommand
- org.jhotdraw.standard.ToggleGridCommand
- org.jhotdraw.figures.GroupCommand
- org.jhotdraw.figures.InsertImageCommand
- org.jhotdraw.figures.UngroupCommand
- org.jhotdraw.util.RedoCommand
- org.jhotdraw.util.UndoCommand
- org.jhotdraw.util.UndoableCommand
- org.jhotdraw.contrib.zoom.ZoomCommand
- Various anonymous classes (not included in the SAIR version of the concern model, since anonymous classes are excluded anyway (see Section 4.4.4)).

**CommandListener**  Role Superimposition sort instance

**Sort context:** Project ‘JHotDraw 6.0b1’

**Role interface:** org.jhotdraw.util.CommandListener

**Imposees:**

- org.jhotdraw.util.UndoableCommand
- org.jhotdraw.util.CommandMenu
- org.jhotdraw.contrib.WindowMenu
- org.jhotdraw.contrib.CTXCommandMenu
- org.jhotdraw.contrib.CTXWindowMenu
Appendix D

Resulting aspects

This appendix lists the aspects as they were created in the case study by the refactoring process presented in Chapter 7. The aspects have not been cleaned up in any way, and the formatting has only been adapted by wrapping the lines if the original aspect could not be fitted on the page.

Virtual roles that were abstracted and are used in the aspects are also shown after the aspect.

D.1 CommandConsistency

This aspect (shown overleaf) contains the refactoring of the CB sort instances of the main command concern. These are the consistent checking of the view (not null) and the consistent notification of the view to check for damage after execution of the command.
package org.jhotdraw.ccconcerns.commands;

import org.jhotdraw.contrib.zoom.ZoomCommand;
import org.jhotdraw.figures.GroupCommand;
import org.jhotdraw.figures.InsertImageCommand;
import org.jhotdraw.figures.UngroupCommand;
import org.jhotdraw.framework.JHotDrawRuntimeException;
import org.jhotdraw.standard.AbstractCommand;
import org.jhotdraw.standard.AlignCommand;
import org.jhotdraw.standard.BringToFrontCommand;
import org.jhotdraw.standard.ChangeAttributeCommand;
import org.jhotdraw.standard.CopyCommand;
import org.jhotdraw.standard.CutCommand;
import org.jhotdraw.standard.DeleteCommand;
import org.jhotdraw.standard.DuplicateCommand;
import org.jhotdraw.standard.PasteCommand;
import org.jhotdraw.standard.SelectAllCommand;
import org.jhotdraw.standard.SendToBackCommand;
import org.jhotdraw.standard.ToggleGridCommand;
import org.jhotdraw.util.RedoCommand;
import org.jhotdraw.util.UndoCommand;

public aspect CommandConsistency {

  pointcut viewNotNullContext_AbstractCommand_execute(
    AbstractCommand abstractCommand) :
    target(abstractCommand) &&
    (within(ChangeAttributeCommand) ||
     within(SelectAllCommand) ||
     within(AlignCommand) ||
     within(CutCommand) ||
     within(ToggleGridCommand) ||
     within(BringToFrontCommand) ||
     within(UngroupCommand) ||
     within(GroupCommand) ||
     within(DeleteCommand) ||
     within(InsertImageCommand) ||
     within(DuplicateCommand) ||
     within(RedoCommand) ||
     within(UndoCommand) ||
     within(CopyCommand) ||
     within(ZoomCommand) ||
     within(PasteCommand) ||
     within(SendToBackCommand)) &&
    execution(void AbstractCommand.execute());

  pointcut postExecutionCheck_AbstractCommand_execute(
    AbstractCommand abstractCommand) :
    target(abstractCommand) &&
    (within(SelectAllCommand) ||
     within(DuplicateCommand) ||
Listing D.1: Aspect implementing main command consistency concerns
D.2 CommandUndoConsistency

This aspect contains the refactoring of the CB sort instances of the command undo concern. These are the initialization of the undo activity and setting the affected figures for this command in the undo activity.

```java
package org.jhotdraw.ccconcerns.commands.undo;
import java.util.List;
import org.jhotdraw.figures.GroupCommand;
import org.jhotdraw.figures.InsertImageCommand;
import org.jhotdraw.figures.UngroupCommand;
import org.jhotdraw.framework.Figure;
import org.jhotdraw.framework.FigureEnumeration;
import org.jhotdraw.standard.AbstractCommand;
import org.jhotdraw.standard.AlignCommand;
import org.jhotdraw.standard.BringToFrontCommand;
import org.jhotdraw.standard.ChangeAttributeCommand;
import org.jhotdraw.standard.CutCommand;
import org.jhotdraw.standard.DeleteCommand;
import org.jhotdraw.standard.DuplicateCommand;
import org.jhotdraw.standard.FigureEnumerator;
import org.jhotdraw.standard.SelectAllCommand;
import org.jhotdraw.standard.SendToBackCommand;
import org.jhotdraw.util.CollectionsFactory;

public privileged aspect CommandUndoConsistency {

    pointcut saveFigureState_DeleteCommand_execute(
        DeleteCommand deleteCommand) :
        target(deleteCommand) &&
        within(DeleteCommand) &&
        execution(void DeleteCommand.execute());

    pointcut saveFigureState_AbstractCommand_execute(
        AbstractCommand abstractCommand) :
        target(abstractCommand) &&
        (within(BringToFrontCommand) ||
         within(GroupCommand) ||
         within(SelectAllCommand) ||
         within(ChangeAttributeCommand) ||
         within(UngroupCommand) ||
         within(AlignCommand)) &&
        execution(void AbstractCommand.execute());

    pointcut saveFigureState_CutCommand_execute(CutCommand cutCommand) :
        target(cutCommand) &&
        within(CutCommand) &&
        execution(void CutCommand.execute());

    pointcut saveFigureState_SendToBackCommand_execute(
        SendToBackCommand sendToBackCommand) :
```
target (sendToBackCommand) &&
within (SendToBackCommand) &&
execution (void SendToBackCommand.execute());

pointcut initUndo_DeleteCommand_execute(
  DeleteCommand deleteCommand) :
  target (deleteCommand) &&
  execution (void DeleteCommand.execute());

before (DeleteCommand deleteCommand):
  initUndo_DeleteCommand_execute (deleteCommand) {
    deleteCommand.setUndoActivity {
      deleteCommand.createUndoActivity();
    }
  }

pointcut initUndo_AlignCommand_execute (AlignCommand alignCommand) :
  target (alignCommand) &&
  execution (void AlignCommand.execute());

before (AlignCommand alignCommand):
  initUndo_AlignCommand_execute (alignCommand) {
    alignCommand.setUndoActivity (alignCommand.createUndoActivity());
  }

pointcut initUndo_BringToFrontCommand_execute (BringToFrontCommand bringToFrontCommand) :
  target (bringToFrontCommand) &&
  execution (void BringToFrontCommand.execute());

before (BringToFrontCommand bringToFrontCommand):
  initUndo_BringToFrontCommand_execute (bringToFrontCommand) {
    bringToFrontCommand.setUndoActivity (bringToFrontCommand.createUndoActivity());
  }

pointcut initUndo_InsertImageCommand_execute (InsertImageCommand insertImageCommand) :
  target (insertImageCommand) &&
  execution (void InsertImageCommand.execute());

before (InsertImageCommand insertImageCommand):
  initUndo_InsertImageCommand_execute (insertImageCommand) {
    insertImageCommand.setUndoActivity (insertImageCommand.createUndoActivity());
  }

pointcut initUndo_UngroupCommand_execute (UngroupCommand ungroupCommand) :
  target (ungroupCommand) &&
  execution (void UngroupCommand.execute());

before (UngroupCommand ungroupCommand):
  initUndo_UngroupCommand_execute (ungroupCommand) {
    ungroupCommand.setUndoActivity (ungroupCommand.createUndoActivity());
  }
D.2 CommandUndoConsistency

Resulting aspects

```java
ungroupCommand.createUndoActivity();

pointcut initUndo_CutCommand_execute(CutCommand cutCommand) :
  target(cutCommand) &&
  execution(void CutCommand.execute());

before(CutCommand cutCommand):
  initUndo_CutCommand_execute(cutCommand){
    cutCommand.setUndoActivity(cutCommand.createUndoActivity());
  }

pointcut initUndo_GroupCommand_execute(GroupCommand groupCommand) :
  target(groupCommand) &&
  execution(void GroupCommand.execute());

before(GroupCommand groupCommand):
  initUndo_GroupCommand_execute(groupCommand){
    groupCommand.setUndoActivity(groupCommand.createUndoActivity());
  }

pointcut initUndo_SendToBackCommand_execute{
  SendToBackCommand sendToBackCommand) :
    target(sendToBackCommand) &&
    execution(void SendToBackCommand.execute());

before(SendToBackCommand sendToBackCommand):
  initUndo_SendToBackCommand_execute(sendToBackCommand){
    sendToBackCommand.setUndoActivity(sendToBackCommand.createUndoActivity());
  }

pointcut initUndo_ChangeAttributeCommand_execute{
  ChangeAttributeCommand changeAttributeCommand) :
    target(changeAttributeCommand) &&
    execution(void ChangeAttributeCommand.execute());

before(ChangeAttributeCommand changeAttributeCommand):
  initUndo_ChangeAttributeCommand_execute{
    changeAttributeCommand){
      changeAttributeCommand.setUndoActivity(changeAttributeCommand.createUndoActivity());
    }

pointcut initUndo_SelectAllCommand_execute{
  SelectAllCommand selectAllCommand) :
    target(selectAllCommand) &&
    execution(void SelectAllCommand.execute());

before(SelectAllCommand selectAllCommand):
  initUndo_SelectAllCommand_execute(selectAllCommand){
    selectAllCommand.setUndoActivity(selectAllCommand.createUndoActivity());
  }
```
pointcut initUndo_DuplicateCommand_execute(
    DuplicateCommand duplicateCommand) :
        target(duplicateCommand) &&
        execution(void DuplicateCommand.execute());

before(DuplicateCommand duplicateCommand):
    initUndo_DuplicateCommand_execute(duplicateCommand)
        duplicateCommand.setUndoActivity{
            duplicateCommand.createUndoActivity();
        }

before(DeleteCommand deleteCommand):
    saveFigureState_DeleteCommand_execute(deleteCommand)
        /*
         * ricardo_padilha: bugfix for correct delete/undelete
         * behavior. When enumerating the affected figures we
         * must not forget the dependent figures, since they are
         * deleted as well!
         */
         FigureEnumeration fe = deleteCommand.view().selection();
         List affected = CollectionsFactory.current().createList();
         Figure f;
         FigureEnumeration dfe;
         while (fe.hasNextFigure()) {
             f = fe.nextFigure();
             affected.add(0, f);
             dfe = f.getDependendFigures();
             if (dfe != null) {
                 while (dfe.hasNextFigure()) {
                     affected.add(0, dfe.nextFigure());
                 }
             }
         }
         fe = new FigureEnumerator(affected);
         deleteCommand.getUndoActivity().setAffectedFigures(fe);

before(AbstractCommand abstractCommand):
    saveFigureState_AbstractCommand_execute(abstractCommand)
        abstractCommand.getUndoActivity().setAffectedFigures(abstractCommand.view().selection());

before(CutCommand cutCommand):
    saveFigureState_CutCommand_execute(cutCommand)
        /*
         * ricardo_padilha: bugfix for correct delete/undelete
         * behavior. When enumerating the affected figures we
         * must not forget the dependent figures, since they are
         * deleted as well!
         */
         FigureEnumeration fe = cutCommand.view().selection();
         List affected = CollectionsFactory.current().createList();
Figure f;
FigureEnumeration dfe;
while (fe.hasNextFigure()) {
    f = fe.nextFigure();
    affected.add(0, f);
    dfe = f.getDependendFigures();
    if (dfe != null) {
        while (dfe.hasNextFigure()) {
            affected.add(0, dfe.nextFigure());
        }
    }
}
fe = new FigureEnumerator(affected);
cutCommand.getUndoActivity().setAffectedFigures(fe);
}
\[before\](SendToBackCommand sendToBackCommand):
saveFigureState_SendToBackCommand\_execute(sendToBackCommand){
    sendToBackCommand.getUndoActivity().setAffectedFigures(
        sendToBackCommand.view().selectionZOrdered());
}
D.3 CommandUndoRole

This aspect contains the refactoring of the RSI sort instance of the virtual undo role. The abstracted virtual role is shown in Listing D.4.

```
package org.jhotdraw.ccconcerns.commands.undo;

import org.jhotdraw.standard.AbstractCommand;
import org.jhotdraw.util.Command;
import org.jhotdraw.util.CommandUndo;
import org.jhotdraw.util.Undoable;
import org.jhotdraw.util.UndoableAdapter;
import org.jhotdraw.util.UndoableCommand;

public aspect CommandUndoRole {

declare parents: Command implements CommandUndo;

private Undoable AbstractCommand.myUndoableActivity;

public Undoable AbstractCommand.getUndoActivity() {
    return myUndoableActivity;
}

public void AbstractCommand.setUndoActivity(
    Undoable newUndoableActivity) {
    myUndoableActivity = newUndoableActivity;
}

public Undoable UndoableCommand.getUndoActivity() {
    return new UndoableAdapter(view());
}

public void UndoableCommand.setUndoActivity(
    Undoable newUndoableActivity) {
    // do nothing: always return default UndoableAdapter
}
}
```

Listing D.3: Aspect implementing undo role for commands

```
package org.jhotdraw.util;

public interface CommandUndo {

    public Undoable getUndoActivity();

    public void setUndoActivity(Undoable newUndoableActivity);
}
```

Listing D.4: The abstracted interface of the ‘UndoableCommand’ role
D.4 CommandObserver

This aspect contains the refactoring of the RSI sort instance of the virtual observer role and
the non-virtual listener role. The abstracted virtual role of the observer is shown in Listing
D.6.

```java
package org.jhotdraw.ccconcerns.commands;

import java.util.EventObject;
import org.jhotdraw.contrib.CTXCommandMenu;
import org.jhotdraw.standard.AbstractCommand;
import org.jhotdraw.util.Command;
import org.jhotdraw.util.CommandListener;
import org.jhotdraw.util.CommandMenu;
import org.jhotdraw.util.ObservableCommand;
import org.jhotdraw.util.UndoableCommand;

public aspect CommandObserver {

decclare parents: Command implements ObservableCommand;
decclare parents: CTXCommandMenu implements CommandListener;
decclare parents: UndoableCommand implements CommandListener;
decclare parents: CommandMenu implements CommandListener;

decclare void AbstractCommand.addCommandListener(
    CommandListener newCommandListener) {
    getEventDispatcher().addCommandListener(newCommandListener);
}

decclare void AbstractCommand.removeCommandListener(
    CommandListener oldCommandListener) {
    getEventDispatcher().removeCommandListener(oldCommandListener);
}

decclare void UndoableCommand.addCommandListener(
    CommandListener newCommandListener) {
    getEventDispatcher().addCommandListener(newCommandListener);
}

decclare void UndoableCommand.removeCommandListener(
    CommandListener oldCommandListener) {
    getEventDispatcher().removeCommandListener(oldCommandListener);
}

/**
 * Description of the Method
 * @param commandEvent
 * Description of the Parameter
 */
decclare void CTXCommandMenu.commandExecuted(
    EventObject commandEvent) {
    // checkEnabled();
}
```
/*
 * Description of the Method
 * @param commandEvent
 * Description of the Parameter
 */
public void CTXCommandMenu.commandExecutable(
    EventObject commandEvent) {
    // checkEnabled();
}

/*
 * Description of the Method
 * @param commandEvent
 * Description of the Parameter
 */
public void CTXCommandMenu.commandNotExecutable(
    EventObject commandEvent) {
    // checkEnabled();
}

public void UndoableCommand.commandExecuted(
    EventObject commandEvent) {
    getEventDispatcher().fireCommandExecutedEvent();
}

public void UndoableCommand.commandExecutable(
    EventObject commandEvent) {
    getEventDispatcher().fireCommandExecutableEvent();
}

public void UndoableCommand.commandNotExecutable(
    EventObject commandEvent) {
    getEventDispatcher().fireCommandNotExecutableEvent();
}

public void CommandMenu.commandExecuted(EventObject commandEvent) {
    // checkEnabled();
}

public void CommandMenu.commandExecutable(EventObject commandEvent) {
    // checkEnabled();
}

public void CommandMenu.commandNotExecutable(
    EventObject commandEvent) {
    // checkEnabled();
}
package org.jhotdraw.util;

public interface ObservableCommand {
    public void addCommandListener(CommandListener newCommandListener);
    public void removeCommandListener(CommandListener oldCommandListener);
}

Listing D.6: The abstracted interface of the ‘ObservableCommand’ role