Automated Software Engineering manuscript No.
(will be inserted by the editor)

An Integrated Crosscutting Concern Migration Strategy and its Semi-Automated Application to JHOTDRAW

Marius Marin · **Arie van Deursen** ·
Leon Moonen · **Robin van der Rijst**

Received: date / Accepted: date

Abstract In this paper we propose a systematic strategy for migrating crosscutting concerns in existing object-oriented systems to aspect-oriented programming solutions. The proposed strategy consists of four steps: mining, exploration, documentation and refactoring of crosscutting concerns. We discuss in detail a new approach to refactoring to aspect-oriented programming that is fully integrated with our strategy, and apply the whole strategy to an object-oriented system, namely the JHOTDRAW framework.

Moreover, we present a method to semi-automatically perform the aspect-introducing refactorings based on identified crosscutting concern sorts which is supported by a prototype tool called SAIR. We perform an exploratory case study in which we apply this tool on the same object-oriented system and compare its results with the results of manual migration in order to assess the feasibility of automated aspect refactoring. Both the refactoring tool SAIR and the results of the manual migration are made available as open-source, the latter providing the largest aspect-introducing refactoring available to date.

We report on our experiences with conducting both case studies and reflect on the success and challenges of the migration process.

Marius Marin
Accenture & Delft University of Technology (Guest), The Netherlands,
E-mail: Marius.Marin@accenture.com

Arie van Deursen
Delft University of Technology, The Netherlands, E-mail: Arie.vanDeursen@tudelft.nl

Leon Moonen
Simula Research Laboratory, Norway, E-mail: Leon.Moonen@computer.org

Robin van der Rijst
Delft University of Technology, The Netherlands, E-mail: rvdrijst@gmail.com

1 Introduction

Software systems change continuously to implement new requirements, to adapt to new architectures and integrate with legacy applications or new systems, or to correct their behavior. The evolution of today's software systems is challenged by their growing size and complexity that stems from the multiple development cycles and changes implemented over time.

1.1 Crosscutting Concerns

The software engineering research and practice has long looked into techniques and tool support to aid software evolution, such as software decomposition and modularization techniques. Most recently, significant attention has been given to the problem of *crosscutting concerns* – application concerns, such as functional or design ones, that do not map onto any single module, e.g., class or method, in a given decomposition of the software system.

The implementation of crosscutting concerns using traditional programming paradigms, such as object-oriented programming, is associated with inherent tangling and scattering problems due to the limitations of their modularization mechanisms. In a typical Java web application, for example, the implementation of concerns like logging or security is scattered over multiple modules and tangled with the core functionality of the module implementing these concerns.

One relatively recent approach to the problem of crosscutting concerns that has gained significant attention is aspect-oriented software development (AOSD)[1] [9]. Aspect-oriented programming techniques introduce several new language constructs and mechanisms that allow to capture crosscutting concerns in dedicated modules, called *aspects*. The modularization of crosscutting concerns helps to more easily localize, understand and modify the implementation of these concerns.

1.2 Migration of Crosscutting Concerns

Adoption of aspect-oriented programming for handling crosscutting concerns in existing systems requires to migrate these systems, and transform the crosscutting implementation of concerns into aspects, a process known as *aspect refactoring*.

Despite significant research efforts on various parts of the refactoring of crosscutting concerns from existing systems, to date there exists no compelling show-case for such a complete migration. One of the main causes for this void is the fact that there is no clearly defined, coherent migration strategy detailing the steps to be taken to perform this process.

Successful migration requires a strategy comprising steps like identification of the concerns (i.e., aspect mining), description of the concerns to be refactored, and consistent application of refactoring transformations. Moreover, such a strategy requires *integrated* migration steps, so that aspect mining results, for example, can be

[1] http://www.aosd.net/

consistently mapped onto concerns in code, and further refactored by general aspect solutions. The present state of the art prevents developers and practitioners from experimenting with a complete migration process and assessing the benefits of migrating to aspect-oriented programming.

In this paper, we propose such an integrated strategy for migrating crosscutting concerns to aspects, which consists of four main steps: (1) idiom-driven identification of crosscutting concerns in an existing system (also known as *aspect mining*); (2) exploration of the concerns identified in the previous step and their context; (3) query-based modeling and documentation of crosscutting concerns in the system; (4) template-based refactoring of the object-oriented idioms into aspect-oriented programming solutions.

Our strategy builds upon the classification and decomposition of crosscutting concerns in so-called *crosscutting concern sorts* that we proposed earlier [21,23]. Each sort describes a typical implementation idiom and relation of crosscutting concerns. Sorts act as glue between the successive steps of the migration: The mining step in our strategy uses the sort-specific idioms to define search-goals for identifying crosscutting concerns that belong to a specific sort (i.e., *sort instances*). To support the exploration and documentation steps, we have formalized the concern sorts using queries over source code and implemented these in a tool for browsing and modeling crosscutting concerns [25].

1.3 Semi-Automatic Aspect Refactoring

While the first three steps of our approach have been covered in our earlier work, this paper focuses on the fourth step and its connection with the three preceding steps. In particular, we define template solutions for the aspect refactoring of instances of each of our sorts.[2] Furthermore, we describe a case study in which we manually apply the whole migration strategy to JHotDraw,[3] an object-oriented application used in other aspect mining and refactoring studies as well [20,5,22,1]. The results of our migration are available as an open-source project called AJHotDraw. This is the largest aspect refactoring publicly available to date that we are aware of.

Finally, we present an approach to semi-automatically perform such aspect-introducing refactorings based on identified crosscutting concern sorts. It is based on having a human in the loop to guide the system through complex refactoring design decisions. The automation of the refactorings to aspects aims at similar benefits as the one currently provided by many development environments for traditional object-oriented refactorings, including automatic code transformations and verifications, or re-use of standard refactoring solutions.

We discuss the implementation of our refactoring approach in a prototype tool called SAIR (Sort-based Aspect-Introducing Refactoring), and conduct an exploratory case study in which we apply this tool on the same object-oriented system as was used for the manual migration. We compare the results of both approaches in order

[2] Our templates target the AspectJ language.
[3] http://jhotdraw.org/

to assess the feasibility and benefits of (semi-)automatic aspect refactoring. SAIR is publicly available to enable repetition experiments and extension by others.

This paper extends our earlier work [24] with (1) an approach to semi-automatically conduct aspect-introducing refactorings, (2) details of the sort specific algorithms for semi-automated refactoring of two sorts: *Role Superimposition* and *Consistent Behavior*, (3) the semi-automatic refactoring tool SAIR, and (4) an exploratory case study in which we apply SAIR to JHOTDRAW and compare its results with the manually migrated AJHOTDRAW.

The remainder of the paper is organized as follows. In Section 2, we give a short introduction into aspect-oriented software development concepts, aspect mining and concern modeling. Then we revisit the previously published notion of crosscutting concern sorts. We describe the migration strategy and elaborate on the first three steps in Section 3. The sort-based aspect refactoring approach that we introduce for the fourth step is presented in Section 4. Section 5 covers our experiences with migrating crosscutting concerns in JHOTDRAW to aspect solutions. Section 6 presents our semi-automatic aspect-introducing refactoring approach, the tool SAIR and our exploratory case study conducted with SAIR on JHOTDRAW. Section 7 presents a second case study that compares both the effort required and aspect-oriented code obtained from a refactoring using SAIR versus a completely manual approach. Section 8 discusses the results and outlines a number of lessons learned. We conclude with an overview of related work and recommendations for future research.

2 Background

In this section, we give a short introduction into the topics of aspect-oriented programming, aspect mining and concern modeling.

2.1 Aspect-Oriented Software Development and AspectJ

Aspect-oriented software development techniques employ a set of new language constructs and composition mechanisms aimed at enhancing the modularization of concerns in software systems. Most of these techniques extend existing object-oriented languages, and particularly Java, using a *join-point model* to attach additional behavior to specific places in the execution of a program.

A *join-point* in this model is a well-defined point in the control flow of a program, such as a method call, a method execution, or a field access (get/set), that can be matched using certain selection criteria. In aspect-oriented programming, join-points are selected using *pointcut* definitions. Figure 1 shows an example of a pointcut in the AspectJ language: The *cmdExecute* pointcut selects the execution of all the `execute()` methods declared in any class in the hierarchy of *AbstractCommand*, except declarations in the *DrawApplication* class. The pointcut designator *this* and the pointcut parameter provide to access the object of the executing method, as we shall see next.

The join-points selected by a pointcut can be attached additional behavior by means of an *advice* construct, as illustrated in Figure 2. The advice specifies the code

```
pointcut cmdExecute(AbstractCommand aCommand) :
  this(aCommand)
  && execution(void AbstractCommand+.execute())
  && !within(*..DrawApplication.*);
```

Fig. 1 An AspectJ pointcut definition.

```
before(AbstractCommand aCommand) : cmdExecute(aCommand) {
  if (aCommand.view() == null) {
    throw new JHotDrawRuntimeException("...");
  }
}
```

Fig. 2 An AspectJ advice.

to be executed automatically when the program reaches the join-point, as well as the moment of the execution, e.g., *before*, *after* or *around* the join-point. The *before* advice in Figure 2 uses the reference to the executing object captured by the pointcut to check a pre-condition and throw a runtime exception.

An aspect-oriented program can declare pointcut and advice in dedicated modules called *aspects*. Aspects are modularization units akin to static classes, which in AspectJ can be declared using the *aspect* keyword.

In addition to the pointcut and advice mechanism, aspects allow for so-called *inter-type declarations* or *introductions* to extend a class or interface with new members or to modify the type hierarchy of a class.

2.2 Aspect Mining

The identification of crosscutting concerns in source code is commonly known as aspect mining. Many of the current aspect mining approaches employ static and dynamic program analysis techniques to search for typical symptoms of crosscutting implementation of concerns, such as code scattering and tangling [20, 2, 5, 31, 16]. Two of the techniques used in this paper, for example, analyze the method call relations in a software system and look for scattered and/or regular patterns of method invocations that are common with many examples of crosscutting concerns such as logging or notification of events [20].

The results of aspect mining consist of program elements and relations that are part of the implementation of a concern. These results can then be used in combination with software browsing and exploration tools to discover the full *extent* of a concern.

2.3 Concern Modeling

Concern modeling aims at enabling multiple decompositions of a software system through methods and tool support that allow for grouping and navigating program elements that belong to the implementation of a certain concern. These concerns are typically not captured in the main decomposition of the system, as they do not map onto dedicated modularization units, such as classes and methods. Concern modeling tools can use dedicated views where the user can associate code elements that implement a specific concern.

Although concern modeling and aspect-oriented programming are similar in many respects, and particularly in the shared goal of improving separation of concerns, there are also several important differences [19]. For example, aspect-oriented programming techniques allow one to ensure that certain behavior is executed consistently for a given set of elements and that this behavior can be enabled or disabled with minimal code changes.

Examples of approaches to concern modeling include concern graphs [29], the Concern Manipulation Environment [14], intentional views [26], and concern sort queries [23]. The latter will be discussed in more detail in Section 3.3 of this paper.

2.4 Crosscutting Concern Sorts

A systematic migration strategy requires a consistent way to address crosscutting concerns in source code. To this end, we distinguish a number of *atomic* crosscutting concerns (i.e., concerns that cannot be split into smaller, still meaningful concerns) that share properties like their implementation idioms and relations. We group concerns that share such properties in categories called *crosscutting concern sorts* [21]. These sorts can be used on their own, but can also be composed to construct more complex crosscutting designs, for example, the *Observer* pattern, often used as a typical example of crosscuttingness.

The first two columns of Table 1 describe the identified sorts and show several examples of instances (the other columns will be introduced in later sections). *Consistent behavior*, for instance, groups concerns whose implementation consists of scattered calls to a specific method implementing the crosscutting functionality. Instances of this sort include, for example, a logging concern, a simple authentication or authorization concern implemented as a call to a method checking credentials, or a mechanism for updating observers using calls to a notification method.

Similarly, the idiom for implementation of secondary roles, common in design patterns like *Observer* or *Visitor*, as well as in mechanisms for persistence, is described by the *Role superimposition* sort.

Composite crosscutting designs exhibit multiple sort instances in their implementation: the aforementioned *Observer* pattern, for example, comprises two instances of *Role superimposition*, for the Subject and the Observer role respectively. Furthermore, it comprises instances of *Consistent behavior*, like the concern for notification of observers, or the one for observers registration. Instances of our sorts are therefore *building blocks* for modeling and describing crosscutting functionality.

Sort and Intent	Examples	Idiom	Template aspect solution

(Method) Consistent Behavior (CB): A set of methods consistently invoke a specific action as a step in their execution.	Logging of exceptions; Wrapping business service exceptions and re-throwing them as new types [20]; Notification of Figure change events.	Method invocations from set of methods.	Pointcut and advice mechanism.
			`around(..) : callersContext(..){`
			`    invokeCB(..); //before`
			`    proceed();`
			`    // or after: invokeCB(..);`
			`}`

Redirection Layer (RL): A type acts as a front-end interface having its methods responsible for receiving calls and redirecting them to dedicated methods of a specific reference, optionally executing additional functionality.	Border decorations for Figure elements (Decorator pattern); Command wrapper for undo support.	Redirector type whose methods consistently forward calls to pair methods in receiver.	Pointcut and around advice replaces redirection.
			`around(..) : call Receiver.m(..) &&`
			`    filteredCallers(..) {`
			`    addBehavior1();`
			`    proceed(..); //redirection`
			`    addBehavior2();`
			`}`

Expose Context (EC): Context Passing: Methods in a call chain consistently use parameter(s) to propagate context information along the chain.	Transaction management [17]; Credentials passing for authorization; Progress monitor for long-running operations.	Method in chain passes parameter as argument to callee.	Pointcut and advice, where the point cut collects the context to be passed - Wormhole [17]
			`around(<caller context>, <callee context>):`
			`    cflow(callerSpace(<caller context>)) &&`
			`    calleeSpace(<callee context>){`
			`    // ... advice body`
			`}`

Role Superimposition (RSI): Types extend their core functionality through the implementation of a secondary role.	Figure elements observed by views for changes (Subject role); Visitable elements (Visitor pattern); Storable Figures (Persistence) [20].	Set of types declare and implement member roles (possibly declared by a distinct interface).	Introduction mechanisms.
			`declare parents : Type implements SecondaryRole;`
			`Modifiers Type Type.roleField;`
			`Modifiers Type Type.roleMethod(..){`
			`    ...//original implementation`
			`};`

Support Classes for Role Superimposition (SC): Types implement secondary roles by enclosing nested support classes. The nesting enforces and specifies the relation between enclosing and support class.	Undo support for Command elements; Event dispatcher for observers' notification.	Set of types (in hierarchy) implement Role using nested classes.	The desired solution, introduction for nested classes, is not supported by AspectJ. Our solution is to move the support classes to the aspect.

Exception Propagation (EP): methods in call chain consistently (re-)throw exceptions from their callees in the absence of an appropriate answer.	IOException thrown if Figure elements recovery fails; Checked SQLException thrown from methods in the JDBC API.	Method in call chain re-throws exception to caller.	Softening exceptions mechanisms.
			`declare soft : ExceptionType :`
			`    (call(* rootException(..)`
			`        throws ExceptionType));`

Table 1 Crosscutting concern sorts.

3 An Integrated Migration Strategy

In this section, we define an integrated strategy for migrating crosscutting concerns in existing systems to aspect-based solutions.

The strategy consists of four steps:

Step 1. Idiom-driven identification of crosscutting concerns (*aspect mining*).
Step 2. Concern exploration.
Step 3. Query-based concern modeling and documentation.
Step 4. Sort-based aspect refactoring.

The remainder of this section discusses the first three steps in more detail and the next section presents the fourth step. We show how the steps are integrated via crosscutting concern sorts using examples from our JHOTDRAW to AJHOTDRAW migration experience.

3.1 Aspect Mining

In our earlier work we have proposed and implemented an idiom-driven approach to aspect mining based on crosscutting concern sorts [22]. The approach supports the design of aspect mining techniques that target instances of a specific sort by searching for the sort's implementation idiom.

The third column in Table 1 shows the implementation idioms associated with each of the sorts. Consider for example the commands in a drawing application, like JHOTDRAW, that carry out tasks in response to user actions. Each command concludes its execution with a call to the `checkDamage` method in the drawing view, which updates the view with changes triggered by the command. The notification concern is an instance of *Consistent behavior* whose implementation idiom is invocation of a specific method from a large set of methods. Aspect mining techniques such as Fan-in analysis [20] or Grouped calls analysis [22] exploit idioms such as this one in their search process.

We have implemented the two mining techniques mentioned above and an additional technique that targets instances of *Redirection layer* in our aspect mining tool FINT[4] [20, 22]. The results of applying FINT to JHOTDRAW are the starting point of our migration case study.

Like the notification mechanism above, we have found the *Consistent behavior* idiom in multiple concerns implementing support for commands and undo operations. Examples include consistently checking the reference to the active view before execution of each command, consistent initialization of Command objects by means of super calls, or consistent checks implemented by all actions to undo a command. Our search for idioms of the *Redirection layer* pointed us to wrapper objects for undo-able commands: methods in the wrapper delegate calls to their wrapped command object.

[4] Available from http://swerl.tudelft.nl/view/AMR/FINT

3.2 Concern Exploration

Aspect mining often does not yield complete crosscutting concern instances, but just concern *seeds*: a set of program elements that belong to a particular crosscutting concern but not necessarily cover the complete concern.

The second step of our strategy, concern exploration, aims at expanding mining results (i.e., concern seeds) to the complete implementation of the associated concerns. In this step, we start from the discovered seeds and use the specific relation of the sort for the seed's concern to identify all the participants in the concern implementation.

In our *Consistent behavior* example, this means looking at all call relations directed to the method `checkDamage` (or another method, depending on the particular concern targeted). As it turns out, not all of the 28 calls to this method that we found are part of the concern of interest, but around two-thirds of them, namely those from *Command* classes. Similarly, the *Grouped calls* mining technique, which applies a more conservative search, covers only partially the set of calls participating in the concern.

A number of tools provide, at least partial, support for exploring seeds and expanding them to full concerns, and for querying source code for concern sort relations. These include FINT [20], the Eclipse IDE, the Concern Manipulation Environment (CME) [33], FEAT [30], JQuery [15], CodeQuest [10], and the Sort Querying Tool SOQUET [25]. The same tools can be used to further understand the context enclosing the discovered crosscutting concern. At this step, we can see, for example, how the identified sort instances in command and undo support relate to each other: commands that can be undone enclose a specialized *UndoActivity* class that knows how to revert the effects of the command's execution. Two of our mined sort instances cover the key methods of the two classes: the `execute` method in a command, and the `undo` one in the enclosed undo activity.

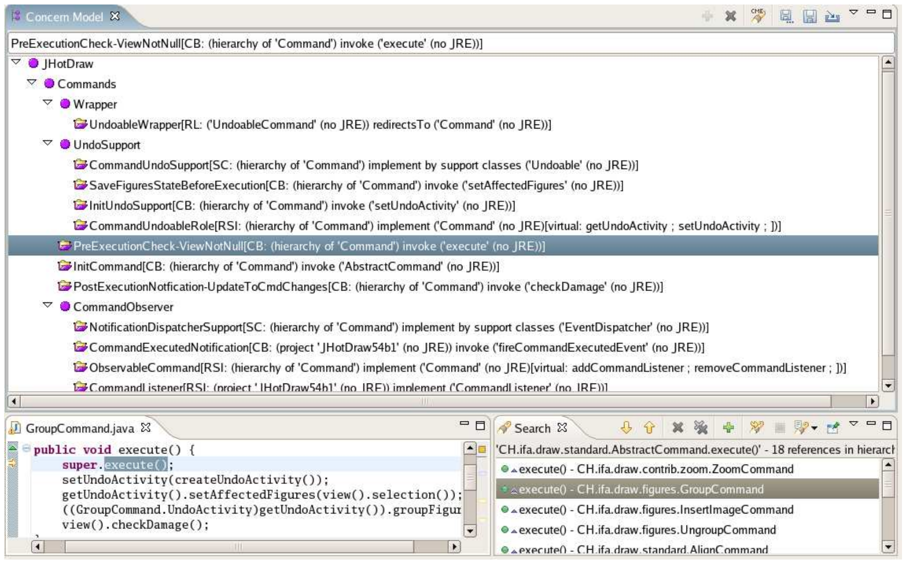
3.3 Concern Modeling and Documentation

Most approaches to concern modeling and their tool support do not enforce consistency across the representation of crosscutting concerns. The decision of what is crosscutting in a system, and how to best represent that, lies with the user of such concern modeling tools. Such a concern model can contain ad-hoc collections of program elements, like methods and classes, that participate in a concern's implementation.

However, to ensure generally applicable solutions for concern migration, we need a coherent way to describe similar concerns and their common properties. To this end, we have defined queries for each of our crosscutting concern sorts which search for the sort's specific relation between source code elements. For more information on these sort queries, we refer to our earlier work [23,19], which formalizes these queries using relation calculus over source models extracted from the system's source code.

We have implemented support for this third migration step in our concern modeling tool SoQueT[5] [25]. Figure 3 shows two of the main views of the tool. The *Con-*

[5] Available from http://swerl.tudelft.nl/view/AMR/SoQueT



```
PreExecutionCheck-ViewNotNull[CB: (hierarchy of 'Command') invoke ('execute' (no JRE))]
JHotDraw
  Commands
    Wrapper
      UndoableWrapper[RL: ('UndoableCommand' (no JRE)) redirectsTo ('Command' (no JRE))]
    UndoSupport
      CommandUndoSupport[SC: (hierarchy of 'Command') implement by support classes ('Undoable' (no JRE))]
      SaveFiguresStateBeforeExecution[CB: (hierarchy of 'Command') invoke ('setAffectedFigures' (no JRE))]
      InitUndoSupport[CB: (hierarchy of 'Command') invoke ('setUndoActivity' (no JRE))]
      CommandUndoableRole[RSI: (hierarchy of 'Command') implement ('Command' (no JRE)[virtual: getUndoActivity ; setUndoActivity ; ])]
    PreExecutionCheck-ViewNotNull[CB: (hierarchy of 'Command') invoke ('execute' (no JRE))]
    InitCommand[CB: (hierarchy of 'Command') invoke ('AbstractCommand' (no JRE))]
    PostExecutionNotfication-UpdateToCmdChanges[CB: (hierarchy of 'Command') invoke ('checkDamage' (no JRE))]
    CommandObserver
      NotificationDispatcherSupport[SC: (hierarchy of 'Command') implement by support classes ('EventDispatcher' (no JRE))]
      CommandExecutedNotification[CB: (project 'JHotDraw54b1' (no JRE)) invoke ('fireCommandExecutedEvent' (no JRE))]
      ObservableCommand[RSI: (hierarchy of 'Command') implement ('Command' (no JRE)[virtual: addCommandListener ; removeCommandListener ; ])]
      CommandListener[RSI: (project 'JHotDraw54b1' (no JRE)) implement ('CommandListener' (no JRE))]
```

```java
public void execute() {
    super.execute();
    setUndoActivity(createUndoActivity());
    getUndoActivity().setAffectedFigures(view().selection());
    ((GroupCommand.UndoActivity)getUndoActivity()).groupFigur
    view().checkDamage();
```

```
'CH.ifa.draw.standard.AbstractCommand.execute()' - 18 references in hierarch
execute() - CH.ifa.draw.contrib.zoom.ZoomCommand
execute() - CH.ifa.draw.figures.GroupCommand
execute() - CH.ifa.draw.figures.InsertImageCommand
execute() - CH.ifa.draw.figures.UngroupCommand
execute() - CH.ifa.draw.standard.AlignCommand
```

Fig. 3 SoQueT documentation of the concerns for Command support in JHotDraw.

cern model view allows us to organize concerns hierarchically, with sort instances and their associated queries as leaf-elements and composite concerns describing more complex crosscutting designs as parents. The user can select a sort instance in the concern model and execute its query; The results of the query are displayed in the *Search (Sorts Result)* view, from where the user can navigate to the corresponding source code. To add a new sort instance to the model, the user launches the dialog providing the query templates for each sort, and parameterizes the query for a given crosscutting concern. For example, to document our *Consistent behavior* instance for notification of views, we use the knowledge gained at the previous steps and search for all the calls to the `checkDamage` method from methods in the *Command* hierarchy. The method and the hierarchy are our input parameters to the query. The instances can then be added to the model from the results view.

A part of the concern model built to document concerns in JHotDraw is shown in Figure 3. The complete model covers over 100 sort instances and is available from the SoQueT web-site.[5] In Section 5, we use this documentation to guide our refactoring and configure the aspect solutions.

4 Aspect Refactoring

We employ a sort-based, idiom-driven approach to aspect refactoring that allows for consistent integration with the previous steps of our migration strategy. Furthermore, we define template aspect solutions for each of our concern sorts that we can instantiate to refactor an occurrence of that sort. Like the previous steps, the refactoring approach addresses crosscuttingness at the level of atomic concerns, which provides the optimal trade-off between complexity of the refactoring and comprehensibility of the refactored element.

The template aspect refactorings for each sort are summarized in the last column of Table 1. A solution basically consist of one aspect language mechanism. However, some sorts do not have an equivalent mechanism in AspectJ or any other aspect language existing at this moment. Support classes, for example, cannot be introduced similarly to role members, although, as we shall see in Section 8, this would be a desired refactoring.

To refactor a sort instance, we start from its query-based documentation in SoQueT. The query points us to the elements participating in the concern, which we use to configure the template aspect solution. For example, the query for a *Consistent behavior* instance indicates the callers to be captured by a pointcut definition (the source context) and the action to be introduced by the advice (the target context). Other configurable elements, such as the type of advice to introduce the crosscutting call (e.g., before, after, after throwing, etc.), are decided at the refactoring time.

The solution described in Table 1 for the *Redirection layer* sort is a common approach to refactoring implementations of the *Decorator* pattern [12, 18]. This consists of replacing the redirector class by an aspect that intercepts relevant calls to the methods receiving the redirection, and then adds the redirector's functionality by means of an advice.

The aspect solution for *Expose context* instances is discussed by Laddad as the *Wormhole* pattern [17]: the extra parameter used to pass context is replaced by using a pointcut to obtain the context from the caller and an advice that makes the context available to the caller's control flow.

Two sorts in table 1 can be refactored using so called *static crosscutting*: the *introduction* and *declare soft* mechanisms of AspectJ are employed in the transformation templates for *Role Superimposition* and *Exception propagation* respectively. As with the other refactorings, the elements to instantiate these templates are available through the sort-based documentation of the concerns. For *Role Superimposition* these are the members of a type's secondary role that have to be moved to, and introduced from, the newly created aspect.

For *Exception propagation*, the template needs to be instantiated with the exception that is propagated. This exception will be wrapped into a, so called, *soft exception* in AspectJ. Unlike normal exceptions, these soft exceptions do not need to be caught or re-thrown. This allows us to remove the *throws* clauses from all transitive callers of the method initiating the exception propagation. The method at the top of the call chain that deals with the exception now has to catch the soft exception which wraps the original checked one. The top method also needs to know the type of the original exception in order to correctly unwrap it. After unwrapping, we no longer need to care about the soft exception and the remaining code to handle the original exception requires no modifications.

5 Case Study I: Manual Aspect Refactoring of JHotDraw

In order to determine the usefulness of the sort-based migration strategy, we conduct an explorative case study, involving the manual migration of selected crosscutting concerns occurring in JHotDraw towards aspects.

5.1 Case Study Design

The questions that we would like to obtain answers to through the case study include:

RQ1: Are the template aspect solutions proposed in Section 4 applicable in practice?
RQ2: What are the risks and benefits of adopting refactoring strategy that is sort-based?
RQ3: Do the refactorings carried out lead to a better separation of concerns?
RQ4: What level of automation of all four steps and the fourth refactoring step in particular is feasible?

The case study is conducted by the first author of this paper. The subject system, JHotDraw, is a drawing framework comprising approximately 20,000 non-comment, non-blank lines of code. It is a system that is frequently used in other aspect mining and refactoring research papers [20, 5, 22, 1].

In the present section, we report on our observations and experiences regarding the manual migration of specific crosscutting concerns towards aspects in JHotDraw. The answers to the questions, as well as an analysis of the threats to their validity are provided in Section 8, where we also discuss the results of our case study concerning the automated refactoring of some of these concerns (covered in Sections 6 and 7).

5.2 AJHotDraw

We share AJHotDraw, the result of our refactoring, and all intermediate refactoring steps as separate versions of an open-source project on sourceforge. [6] To our knowledge, this is the largest migration to aspects available to date.

A transparent, gradual migration process is important for building confidence in the aspect-oriented solution. Therefore, our refactorings aim at maintaining the conceptual integrity and stay close to the original design. In addition, by sharing the concern model and publishing the refactoring steps in a versioned repository, we aim provide insight in the migration process and enable traceability, making the refactored system easier to understand.

The discussion below focuses on the refactoring of sort instances contained in the implementation of the command and undo functionality, which we also used in Section 3 to explain the first three steps of the approach. We use the organization of concerns in the concern model initiating the refactoring to design the package and type structure of our aspect solutions. The solutions discussed below have been integrated with the code in the public repository.

[6] http://sourceforge.net/projects/ajhotdraw/

5.3 Consistent Behavior in Command

JHOTDRAW makes use of the *Command* design pattern in order to separate the user interface from the underlying model, and to support such features as undoing and redoing user commands. Each command has to realize the *Command* interface, for which a default implementation is provided in the *AbstractCommand* class. The key method is *execute*, which takes care of actually carrying out the command (such as pasting text, duplicating a figure, inserting an image, etc.).

A typical implementation of a command is highly crosscutting, with the *Command* top interface defining three different roles: besides their core functionality, commands are undo-able as well as observable elements. The support for the secondary roles counts for half of the *Command*'s members. Similarly, the `execute` method in a typical concrete command implements multiple concerns.

Each `execute` method should start with a consistency check verifying that the underlying "view" exists. Therefore, each concrete implementation of `execute` starts with a call to the `execute` implementation in the superclass, which is always the one from the *AbstractCommand*. This is illustrated in Figure 4.

We apply a *Consistent behavior* refactoring template from the last column in Table 1 using a pointcut capturing all `execute` methods, and putting the check itself in the advice. Observe that mimicking the implementation where the check is in a super method is not possible in AspectJ: super methods cannot be accessed when advising a method. The resulting solution is shown in Figure 5.

```
public class AbstractCommand implements Command {
  ...
  public void execute() {
    if (view() == null) {
      throw new JHotDrawRuntimeException(
        "execute should NOT be getting called when" +
        "view() == null");
} } }

public class PasteCommand extends AbtractCommand {
  ...
  public void execute() {
    super.execute();
    ...
} }
```

Fig. 4 Consistent check - super method idiom.

```
pointcut cmdExecute(AbstractCommand aCommand) :
  this(aCommand)
  && execution(void AbstractCommand+.execute())
  && !within(*..DrawApplication.*);

before(AbstractCommand aCommand) : cmdExecute(aCommand) {
  if (aCommand.view() == null) {
    throw new JHotDrawRuntimeException("...");
} }
```

Fig. 5 Enforcing consistency using advice.

The only surprise in this code may be the `within` clause in the pointcut. In the exploration step, we learned that *anonymous* subclasses of *AbstractCommand* do not implement the consistency check. Such classes are used for simple commands like printing, saving, and exiting the application. Since AspectJ does not provide a direct way to exclude anonymous classes in a pointcut, we used the `within` operator to exclude executions occurring in the context of the top level object creating the full user interface. One can also argue that the anonymous classes should include this check in which case the exclusion can be omitted from the pointcut. However, as stated before, we focus on keeping the behavior as it was, not on modifying it.

Besides the separation of the consistency check from the core logic of the commands, another benefit of the aspect approach is that consistency checks cannot be forgotten. This is illustrated by a number of the anonymous classes, but also by one non-anonymous command,[7] which does not extend the *AbstractCommand* default implementation. Consequently, it cannot reuse the consistency check using a supercall. Inspection of the `execute` implementation, however, clearly shows that the code exits with a null pointer exception in case the check fails. This suggests that the aspect that we are looking for should implement the check not only for the *AbstractCommand* class, but for all the *Command* implementations.

5.4 Undo Functionality

Support for “undo” functionality was added in JHotDraw version 5.4. As can be imagined, it is a concern that cuts across many different classes. More than 30 elements of the JHotDraw framework, comprising *commands*, *tools* and *handles*, have associated undo constructs to revert the changes spawned by their underlying activities. The *commands* group is the largest in terms of defined undo activities.

The participants of the “Undo” functionality have the following responsibilities:

– Each command is associated with one *undo activity*, whose method *undo* can be invoked to revert the command. The undo activity is implemented in a nested class of the command, which is instantiated using a factory method called `createUndoActivity`.
– Prior to the execution of the command’s core logic, the command saves a reference to its associated undo activity, by calling a dedicated setter method.
– The primary abstraction in the undo activity is the list of affected figures: when the command’s `execute` method is invoked, the relevant state of the affected figures is stored in the undo activity.
– Undo activities are maintained on a stack by the undo manager.

5.4.1 Support Classes for Role Superimposition

The refactoring that we propose for Undo consists of associating a dedicated undo-aspect to each undo-able command. The aspect implements the entire undo functionality for the given command, while the associated command class remains oblivious to its secondary (undo) concern.

[7] Namely, the *UndoableCommand*.

We use naming conventions to relate the aspect to its supported command class. In a successive step, we refactor each of the sort instances in the undo support. The command's nested *UndoActivity* class belongs to a *Support classes* instance. In the absence of introduction mechanisms for nested classes in AspectJ, our aspect solution consists of moving the *UndoActivity* class into the aspect.

The factory methods for the undo activities (`createUndoActivity()`), as well as the members for managing the reference to the command's undo activity belong to an instance of *Role superimposition*. The role members move to the aspect, from where they are introduced back into the associated command classes using inter-type declarations. The design, however, suffers modifications as the visibility of the undo factory methods has been altered: ASPECTJ cannot be used to introduce the required factory method as *protected*.

5.4.2 Consistent Behavior

The invocations in the `execute` method that are responsible for setting up the undo activity implement *Consistent behavior* concerns: the calls are taken out of the `execute` method, and woven into it by means of advice. In some cases the corresponding pointcut simply needs to capture all `execute` method calls. However, in other cases the pointcut is more complex, depending on the way the undo code is mixed with the regular code.

As an example to illustrate that automating such refactorings is not at all straightforward, consider the paste-command, whose `execute` method consists of retrieving the selected figures from the clipboard, inserting them into the current view, and clearing the clipboard. All this is done in a single method, using local variables and if-then-else statements to deal with situations like pasting from an empty clipboard. The undo aspect will require the same conditional logic, and access to the same data in the same order. The following alternatives are possible for aspect refactoring:

– if all getters are side effect free, an approach is to setup the undo activity in a simple before advice. In JHOTDRAW, however, this is not the case, for example because of figure enumerators that have an internal state.
– an alternative is to intercept relevant getters, keep track of the data locally in the advice as well, and inject advice after all data has been collected. This is the approach we follow, but some of the pointcuts are somewhat artificial. Figure 7 shows such a pointcut in the undo aspect for the *PasteCommand*, refactored from Figure 6. The *clipboardGetContents()* pointcut captures the call that sets the reference to be checked by both the command's core logic and the undo functionality in the aspect.
– The last possibility is to refactor the long `execute` method into smaller steps using non-private methods. The extra method calls can be intercepted allowing smooth extension with setting up the undo activity, at the cost of creating a larger interface and breaking encapsulation. Moreover, we would still introduce artificial pointcuts, as our intention is to enhance the behavior of the `execute` method, and not of various steps created for supporting advice introduction.

```
public class PasteCommand extends FigureTransferCommand {
  public void execute() {
    ...
    FigureSelection selection = (FigureSelection)
        Clipboard.getClipboard().getContents();
    if (selection != null) {
        setUndoActivity(createUndoActivity());
        ... //core command logic and other undo setup
        FigureEnumeration fe = insertFigures(...);
        getUndoActivity().setAffectedFigures(fe);
        ...
}   }   }
```

Fig. 6 The original PasteCommand class.

```
public aspect PasteCommandUndo {
  //store the Clipboard's contents - common condition
  FigureSelection selection;

  pointcut clipboardGetContents() :
      call(Object Clipboard.getContents()) &&
      withincode(void PasteCommand.execute());

  after() returning(Object select):clipboardGetContents(){
          selection = (FigureSelection)select;
  }
  ...

  pointcut executePasteCommand(PasteCommand cmd) :
      this(cmd) && execution(void PasteCommand.execute());

  // Execute undo setup
  void after(PasteCommand cmd):executePasteCommand(cmd) {
      // the same condition as in the advised method
      if(selection != null) {
          cmd.setUndoActivity(cmd.createUndoActivity());
          ...
          cmd.getUndoActivity().setAffectedFigures(...);
}   }   }
```

Fig. 7 The undo aspect for PasteCommand.

5.4.3 Redirection Layer

The design of undo in JHotDraw uses wrapper objects to associate undo-able commands to menu items and buttons in the user interface (UI). The wrappers share their top level interface with regular commands, so they can connect to UI elements and receive user actions. While most commands are undo-able and wrapped by an *UndoableCommand* object, there are a few exceptions, such as *CopyCommand*.

Wrappers are instances of *Redirection layer*. The refactoring of such instances raises several important issues: first, we need to identify those commands that are wrapped by an *UndoableCommand* object and accessed through this object; second, we need to check if all clients of a command access its functionality via the wrapper. Only those calls from command clients that are received by a wrapper in the original implementation need to be captured by the aspect solution to attach the wrapper's functionality by means of advice.

Further complications that limit feasibility of refactoring automation have to do with the multiple roles in *UndoableCommand*: since the aspect solution completely replaces the wrapper class, this means that introduction of roles is no longer possible. Some of the original roles in the system are implemented by the wrapper only to comply with the top interface of the wrapped element and add no specific functionality, such as the *Observable* role of *Command*s. The aspect solution can safely omit these roles. For other roles however, this is not desired and refactoring requires customized redirector solutions.

6 Semi-automatic Refactoring towards Aspects

The refactorings presented in the previous section show promising results for the automation of our concern sort-based approach, as we have been able to reuse the proposed refactoring templates for various sort instances. However, we notice a number of challenges that need to be considered by an automatic or semi-automatic refactoring solution. For example, the description of a crosscutting concern by a query in SoQueT and the refactoring template that gives the AspectJ mechanism to be used for modularizing the concern, do not always suffice for a successful migration of the concern's implementation. The crosscutting element to be moved to an aspect by refactoring might access members that are not visible from the aspect, and hence the migration would result in a compilation error. While in a manual approach these issues are typically easy to fix, it is desirable that any partially automated solution is able to detect them and to assist the user in addressing them.

Due to the detailed knowledge that is required to solve migration subtleties like the one sketched above, a completely automated approach is next to impossible to achieve. Instead, we will focus an a semi-automated approach, with a human in the loop to guide the system through the right decisions.

In the remainder of this section, we set out to design and implement tool support for such a semi-automated aspect-introducing refactoring. We will focus on two of our concern sorts: *Consistent Behavior* and *Role Superimposition*. These two sorts are particularly relevant for investigating automation as they are the most commonly encountered sorts in practice, including in the JHotDraw case [23]. Moreover, these sorts are supported by most of the current aspect mining techniques [19]. Finally, each of them requires a different language mechanism for its refactoring to AspectJ, namely pointcut-and-advice, and introduction, respectively.

6.1 Approach

The automation of the refactoring process can be devised in five main phases:

Concern input: Describe the concern to be refactored as a sort instance, and map the elements in the description of the concern into the refactoring template of that sort. For example, the call sites in a *Consistent behavior* instance will be mapped onto the pointcut definition, while the method invoked consistently will give us the action to be implemented by the advice.

Target selection: Select the target of the migration, i.e., the aspect module to which the sort instance will be migrated.

Solution configuration: Analyze the mapping of the concern description onto the refactoring template and collect information for configuring the aspect solution if the mapping is not perfect. For example, certain concerns provided as input for refactoring at the first step might require additional configuration, like the type of advice to replace a *Consistent behavior* call. This could be the case when the position of the call in the caller's body does not indicate a clear before/after advice, but the call might well be turned into such an advice. Yet, such a decision needs to be provided as an input in the refactoring.

Challenge detection and resolution: Analyze the impact of the migration, report on possible challenges, which could result in compilation errors after migration, and provide a default set of alternative fixes to these challenges. Examples of challenges include changes to member access from the code to be migrated to aspects, or references via the *super* keyword that cannot be preserved by the migration.

Concern migration: Apply code transformations to extract the crosscutting implementation of the concern into the aspect solution.

6.2 Algorithm

In order to illustrate the approach, the algorithm for refactoring instances of *Consistent Behavior* to an aspect-based solution is shown in Algorithm 1. The algorithm consists of the following parts:

lines 6-8: The information to create the pointcuts and advice is determined.

lines 11-15: If pointcut grouping was requested, the groupable pointcuts are determined. For each group, one pointcut and one advice is created.

lines 17-23: If advice grouping was requested, the pointcuts that can be grouped are determined. For each context method a pointcut is created, but for each *group*, one advice is created.

lines 25-28: If no grouping was requested, one pointcut and one advice is created for each context method.

lines 31-33: Finally, the consistent calls are removed from the context since they are now in the advice.

6.3 Implementation

Our refactoring support is implemented as an Eclipse plug-in called SAIR, a tool to support *Sort-based Aspect-Introducing Refactoring*. The tool is freely available,[8] together with a detailed description of the implementation [28].

The tool's architecture consists of components designed to address each of the steps in the migration process outlined above. The gathering of the input is realized by accessing the concern description in SoQueT, exposed via a public interface and

[8] http://swerl.tudelft.nl/view/AMR/SAIR

Algorithm 1 Migrating CB sort instances

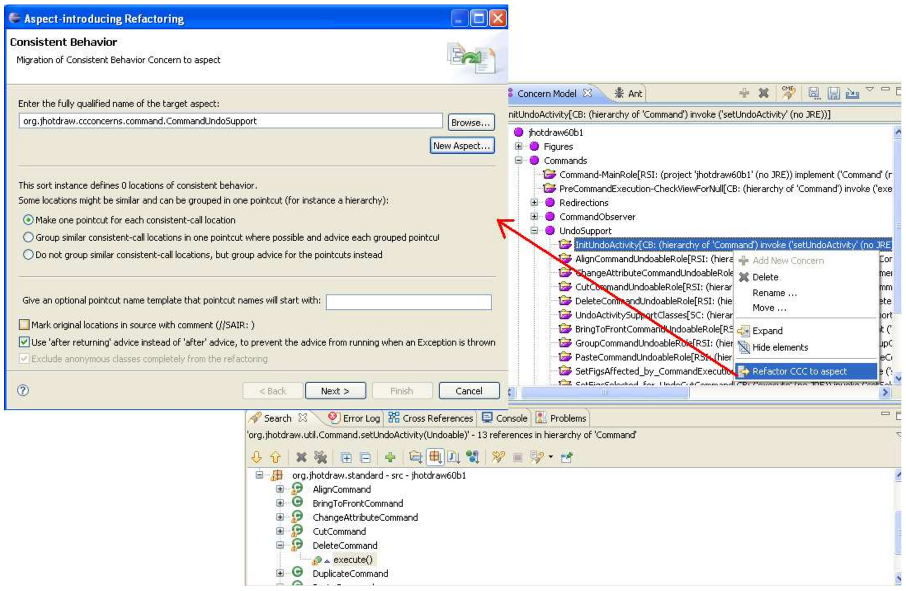
```
1:  procedure PERFORMMIGRATION($input$)
2:      $sort \leftarrow input.sort$
3:      $call \leftarrow sort.consistentCall$
4:      $aspect \leftarrow input.targetAspect$
5:      $contexts \leftarrow sort.contextMethods$
6:      for all $contexts$ as $context$ do
7:          $pcinfos[] \leftarrow$ DETERMINEPOINTCUTINFO($context$)
8:      end for
9:
10:     if $input.groupPointcuts$ then
11:         $grouped \leftarrow$ GROUPPOINTCUTS($pcinfos$)
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:         $grouped \leftarrow$ GROUPPOINTCUTS($pcinfos$)
18:         for all $grouped$ as $group$ do
19:             for all $grouped.pcinfos$ as $pointcutinfo$ do
20:                 CREATEPOINTCUT($group$)
21:             end for
22:             CREATEGROUPEDADVICE($group, aspect$)
23:         end for
24:     else
25:         for all $pcinfos$ as $pointcutinfo$ do
26:             CREATEPOINTCUT($pointcutinfo, aspect$)
27:             CREATEADVICE($pointcutinfo, aspect$)
28:         end for
29:     end if
30:
31:     for all $contexts$ as $context$ do
32:         REMOVECALLFROMCONTEXT($context, call$)
33:     end for
34: end procedure
```

the definition of "extension points" in Eclipse. For the current version of SAIR, we use a customized version of SOQUET that provides SAIR with the description of a concern by its extent, namely the sets of crosscut and crosscutting elements. If we consider an instance of *Consistent Behavior*, such as notification of observers, the description of the concern's extent passed to SAIR is as follows:

$$CB : \{\{m_1, ..., m_i, ...m_n\}, \{notifyObservers(..)\}\} \quad (1)$$

where, the first set in the description gives the caller-methods that consistently invoke the notifier, and the second set consists of the notification method. SAIR is also aware of the sort of the specific concern to be refactored, and hence, of the crosscutting relation, which is specific to each sort.

Furthermore, SAIR implements a set of user interface "wizards" for collecting information from the user, such as the target aspect for migration. The integration with SOQUET and the wizard page are shown in Figure 8: the refactoring option can be activated from the *Concern Model* view of SOQUET, by selecting it in the context



Consistent Behavior

Migration of Consistent Behavior Concern to aspect

Enter the fully qualified name of the target aspect: org.jhotdraw.ccconcerns.command.CommandUndoSupport

This sort instance defines 0 locations of consistent behavior.
Some locations might be similar and can be grouped in one pointcut (for instance a hierarchy):

- [✓ Make one pointcut for each consistent-call location]
- Group similar consistent-call locations in one pointcut where possible and advice each grouped pointcut
- Do not group similar consistent-call locations, but group advice for the pointcuts instead

Give an optional pointcut name template that pointcut names will start with:

- Mark original locations in source with comment (//SAIR:)
- [✓ Use 'after returning' advice instead of 'after' advice, to prevent the advice from running when an Exception is thrown]
- [✓ Exclude anonymous classes completely from the refactoring]

```
nitUndoActivity[CB: (hierarchy of 'Command') invoke ('setUndoActivity' (no JRE))]
jhotdraw60b1
  Figures
  Commands
    Command-MainRole[RSI: (project 'jhotdraw60b1' (no JRE)) implement ('Command' (r
    PreCommandExecution-CheckViewForNull[CB: (hierarchy of 'Command') invoke ('exe
    Redirections
    CommandObserver
    UndoSupport
      InitUndoActivity[CB: (hierarchy of 'Command') invoke ('setUndoActivity' (no JRE)
      AlignCommandUndoableRole[RSI: (hiera
      ChangeAttributeCommandUndoableRole
      CutCommandUndoableRole[RSI: (hierar
      DeleteCommandUndoableRole[RSI: (hie
      UndoActivitySupportClasses[SC: (hierar
      BringToFrontCommandUndoableRole[RS
      GroupCommandUndoableRole[RSI: (hier
      PasteCommandUndoableRole[RSI: (hier
      SetFigsAffected_by_CommandExecutio
```

```
'org.jhotdraw.util.Command.setUndoActivity(Undoable)' - 13 references in hierarchy of 'Command'
org.jhotdraw.standard - src - jhotdraw60b1
  AlignCommand
  BringToFrontCommand
  ChangeAttributeCommand
  CutCommand
  DeleteCommand
    execute()
  DuplicateCommand
```

Fig. 8 SAIR's integration with SOQUET and user interface for input collection.

menu of the sort-instance that documents the crosscutting concern to be refactored. This action opens the wizard for configuring the refactoring solution.

The bottom of the figure shows the *Search* view, which displays the results of the query documenting the selected concern. The query can be repeated, e.g., to account for possible changes in the code, by selecting the *Expand* option from the context menu.

The *solution configuration* step is supported through a set of wizard pages, some of which are specific to each sort. The configuration of the refactoring for *Consistent Behavior* instances allows the user to specify the type of advice to apply (e.g., after, after returning, etc.). In addition it allows for configuration of the generated pointcut, which includes options for grouping the call sites in one or multiple pointcuts.

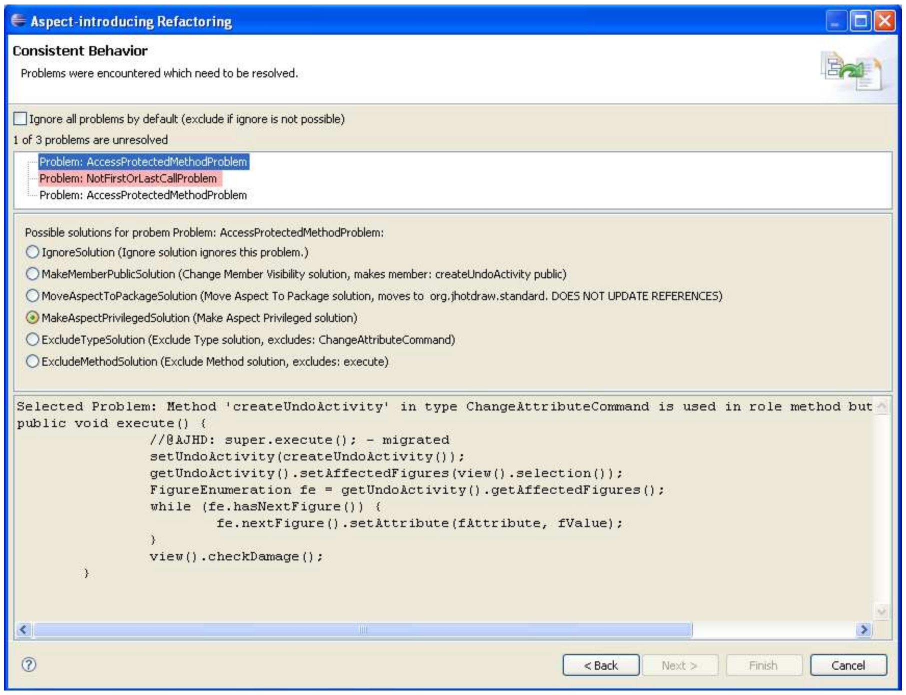
Likewise, *Role superimposition* might require configuration for so-called *virtual* roles. SOQUET allows the user to define a virtual role for documenting a set of members in a type that implement a distinct role, but which are not declared by a dedicated interface. Besides migrating the members making up a virtual role to ASPECTJ inter-type declarations, SAIR also provides options for extracting the role members into an interface declaration.

After configuring the solution, the component for *challenge detection and resolution* analyzes the impact of the migration, before applying any code changes. Table 2 shows a selection of challenges that are checked and the set of possible solutions for each challenge proposed to the user. The set of choices also includes the exclusion of some elements in the concern description that will, then, not be considered for refactoring. Figure 9 illustrates the dialog for challenge resolution, with a list of detected

Table 2 Challenge detection and resolution.

Challenge	**Resolution**[*]
Visibility member enclosing class	Use privileged access; Modify visibility of member; Add public accessor
Visibility member enclosing package	Use privileged access; Modify visibility of member; Add public accessor; Move aspect to package
Visibility member nested class	Use privileged access; Modify visibility of member
Visibility member package visible nested class	Use privileged access; Modify visibility of member; Move aspect to package
Access local variable	Turn local variable into aspect field
References using *super*	Add public accessor; Migrate super body; Create redirection to super
Advise on inner code fragment	Extract method for inner fragment and advise as before or advise as after on extracted method

[*] An additional resolution approach common to all challenges is the option to exclude the offending element from the list of elements-to-refactor.



Consistent Behavior
Problems were encountered which need to be resolved.

[ ] Ignore all problems by default (exclude if ignore is not possible)
1 of 3 problems are unresolved

Problem: AccessProtectedMethodProblem
Problem: NotFirstOrLastCallProblem
Problem: AccessProtectedMethodProblem

Possible solutions for probem Problem: AccessProtectedMethodProblem:
- ○ IgnoreSolution (Ignore solution ignores this problem.)
- ○ MakeMemberPublicSolution (Change Member Visibility solution, makes member: createUndoActivity public)
- ○ MoveAspectToPackageSolution (Move Aspect To Package solution, moves to org.jhotdraw.standard. DOES NOT UPDATE REFERENCES)
- [✓ MakeAspectPrivilegedSolution (Make Aspect Privileged solution)]
- ○ ExcludeTypeSolution (Exclude Type solution, excludes: ChangeAttributeCommand)
- ○ ExcludeMethodSolution (Exclude Method solution, excludes: execute)

```
Selected Problem: Method 'createUndoActivity' in type ChangeAttributeCommand is used in role method but
public void execute() {
                //@AJHD: super.execute(); - migrated
                setUndoActivity(createUndoActivity());
                getUndoActivity().setAffectedFigures(view().selection());
                FigureEnumeration fe = getUndoActivity().getAffectedFigures();
                while (fe.hasNextFigure()) {
                        fe.nextFigure().setAttribute(fAttribute, fValue);
                }
                view().checkDamage();
        }
```

Fig. 9 Challenge resolution in SAIR.

challenges and a number of options for the selected one. After all the challenges are resolved, the tool provides a preview of the changes.

The last step consists of code transformations that remove the crosscutting implementation and move it to an ASPECTJ module. Although the AspectJ Development

```
public class DeleteCommand implements Command {
   public void execute(){
      // @SAIR: super.execute(); moved to advice in aspect
      // org.jhotdraw.ccconcerns.commands.CommandContracts
      setUndoActivity(createUndoable());
      // ... [core logic to delete a selection of figs] ...
   }
   // ... [other Command methods] ...
}
```

Fig. 10 Consistent behavior instances in JHotDraw's DeleteCommand class.

Tools for Eclipse[9] provide support for parsing aspects, they currently do not support aspect code generation for inter-type declarations, pointcuts and advice. As a result, the largest part of this code generation had to be implemented from scratch, without library support.

7 Case Study II: Refactoring JHOTDRAW using SAIR

In order to assess the usefulness of semi-automatic refactoring using SAIR, we set up a second case study, which again was explorative in nature. This case study aims to answer the following questions:

RQ5: How does the aspect-oriented code obtained through SAIR compare to code obtained via a manual refactoring process?
RQ6: How does the manual effort involved in the use of SAIR compare to a completely manual refactoring approach?

This second case study was conducted by the fourth author of this paper, who was not involved in the original manual refactoring of JHOTDRAW.

To conduct the case study, we selected a set of concerns from JHOTDRAW that were also manually refactored. The aspect-based implementation of these concerns in AJHOTDRAW, together with the set of unit tests to ensure behavior preservation, give us the proper base for assessing our approach to automatic refactoring and for comparing the results of SAIR with the ones of the manual refactoring.

Figure 10 shows the undo-setup concern to be refactored using SAIR. The concern is an instance of *Consistent behavior* cutting across the `execute` methods in *Command* classes. The first commented lines of code show the implementation of a concern that has already been refactored. The comments are generated automatically for each instance being migrated. The results of the automatic migration are illustrated in Figure 11.

The aspect implementation of the concern is fairly simple and consists of multiple pointcuts that capture the execution of the context methods, and a *before* advice for the consistent call to be applied at each captured joinpoint. The simple pointcuts could also be grouped in composed ones. However, the current version of the tool does not support more complex features like abstraction or wildcards.

[9] AJDT - http://www.eclipse.org/ajdt/, v.1.4.2

```
public aspect CommandUndoSupport {
   pointcut deleteCommand_execute(DeleteCommand deleteCommand) :
      target(deleteCommand) &&
      execution(void DeleteCommand.execute());

   before(DeleteCommand deleteCommand): deleteCommand_execute(deleteCommand){
      deleteCommand.setUndoActivity(deleteCommand.createUndoActivity());
   }

   pointcut changeAttributeCommand_execute(ChangeAttributeCommand
      changeAttributeCommand) :
   //...
}
```

Fig. 11 Aspect implementation of consistent behavior

The SAIR-aided migration targeted eight sort instances from the set of concerns for Command support refactored in AJHOTDRAW. Seven of these instances could be successfully migrated, while one instance of *Consistent behavior* implemented by means of calls to the *super(..)* constructor could not be handled by the tool (nor by AspectJ, as discussed earlier).

Two of the successful refactorings also requried small manual adjustments in order to improve on the final result; for example, certain declarations exclusively used by the migrated code were moved manually to the aspect solution. In addition to this, we excluded a few highly tangled elements from the set of elements to be refactored for specific concerns using the *exclusion* option in SAIR. In most of these cases, the tangling could have been reduced by a preliminary restructuring of the original JHOTDRAW source code for the refactored methods.

A step that also currently requires a manual solution consists of the ordering of advice applying at the same joinpoints. This was the case for instances of *Consistent behavior* originally implemented by successive method calls.

8 Discussion

In this section we reflect on our experience with migrating crosscutting concerns to aspect-oriented programming and discuss the main challenges and limitations encountered in practice. The discussion covers general considerations of the refactoring to aspects as well as considerations specific to our proposed approach. The discussion is based on the research questions posed in our two case studies. Furthermore, we include an analysis to the threats to validity involved in our cases.

8.1 RQ1: Applicability in Practice

The proposed template aspect solutions proved suitable for refactoring concrete sort instances in the JHOTDRAW case and for separating the crosscutting code from the core system. However, the difficulty of implementing the aspect solution and the quality of the result will vary from case to case. One of the issues is pointcut definitions:

Table 3 Risks and possible limitations of the aspect solution.

Sort	**Limitations and risks**
Consistent Behavior	Advice constructs in a privileged aspect can break encapsulation; High degree of tangling might prevent automatic refactoring; Anonymous classes cannot be referred to consistently, preventing generic pointcuts; Calls to super class functionality cannot be migrated into advice; Modular reasoning affected by need to keep track of data set in the advised method; Check required that omissions are not on purpose; Sophisticated pointcuts needed to intercept all relevant state modifications in the advised methods; Check required that precedence does not change due to new advice;
Redirection layer	The repetitive logic of redirection for the redirector's methods is not eliminated – the aspect solution addresses the redirection at method level and not at type level; New redirector methods are not automatically covered by the solution; The aspect solution is not dynamic (dynamic reordering of redirectors) [12]; The aspect solution replaces the redirector (wrapper) and hence changes the public interface of the application to test against; The calls to the receiver to be advised for redirection need to be detected;
Role superimposition	Visibility affected since protected (/non-public) methods cannot be introduced.
Support classes for role superimposition	Not supported; Nesting the support class in the aspect breaks dependencies (thus forcing the enclosing class to make more of its interface public) and weakens the relation with the enclosing class;
Exception propagation	Type of thrown exception is lost; Refactoring *throws* clauses in inheritance hierarchy.

Ideally, we would like to use pointcut definitions that describe a set of elements by formalizing a common property instead of a brittle enumeration of the elements in the set. In practice, such definitions will not always be feasible, either because of limitations in the aspect language, or due to irregularities in the code under investigation.

Desired functionality included for example a pointcut to capture calls from "all *Command* classes, except all anonymous classes", or a pointcut for "all objects interested in command events". Irregularity in the code might require that for certain methods the advice executes only if a specific condition holds. This is the case for a few commands in JHotDraw that send notifications of their execution only if the clipboard's content is not empty. In such a situation, one has to make a trade-off between a generic pointcut definition that captures all commands, but ignores the particular condition, and a definition that enumerates all appropriate elements. The former solution would execute the code in the advice in spite of its void effect. However, the latter pointcut definition needs to be updated manually for every new element that is added to the set of interest (i.e., every new command).

Similar observations can be made about the definition of advices: sometimes we need to modify the original control flow of a method-to-be-migrated in order to introduce an action to it by means of advice. Although the refactoring may have no effect on the observable behavior of the method, the original flow could be more natural or comprehensible.

8.2 RQ2: Risks and benefits of sort-based refactoring

In comparison with refactoring approaches proposed by others, our sort-based migration strategy gives a clear definition of the input required for refactoring (i.e., an atomic concern) and describes it consistently using queries. This allows for the definition of reusable solutions and improves comprehension of refactoring by addressing meaningful concerns instead of code fragments [1, 27]. Moreover, the concern queries allow us to describe the context cut across by a concern, and hence the concern's intent. This gives a better insight into the concern and its aspect solution than the simple enumeration of joinpoints common with most previous refactoring approaches. We believe that a clearly specified input for a refactoring solution is a necessary condition for ensuring consistent migration of concerns.

Among the main risks of refactoring, we identify the high level of coupling and complex dependencies between the base code and the crosscutting concern. We anticipate that any non-trivial aspect refactoring will require object-oriented refactorings, before the crosscutting concern can be taken out of the available system.

The issue with coupling is that, before migration, concern code can freely access certain parts of the core code that may have limited visibility after the migration. Possible risks in such a case are weakening the visibility restrictions of those members or violating encapsulation by declaring the aspect *privileged*. Other risks include code duplication in advice and the advised method or definition of artificial pointcuts to capture return values of calls from the advised method. This could be the case when some control logic is required by both aspect and the advised method.

We encountered several complex dependencies while refactoring instances of *Exception propagation* in JHotDraw. One example is the propagation of the *IOException* rooted in the set of methods to read drawings from file. The methods in the call chain re-throwing the exception override other methods, whose declared thrown exceptions might only serve for compliance with the method to be refactored. In this case, we also need to address their *throws* clause within our refactoring. Moreover, the overriding elements of a method in the chain that throw the same exception need to be refactored too, as their exception declaration is no longer allowed.

Table 3 provides a more complete overview of the risks and limitations in refactoring to aspects. Note that many of these limitations are independent of the strategy employed for refactoring. In spite of that, we are not aware of other papers in the area of refactoring to aspects that discuss these limitations.

The subject of our case study is a framework for drawing applications that was developed as a showcase of design pattern use. Alternatively, most of the examples of AspectJ aspects use web applications as their application domain. This raises the question if the current aspect implementations are more suitable for certain application domains and crosscutting concerns than for other? Based on the evidence we collected thus far, we conjecture that this indeed could be the case. This also suggests a further advantages of having templates and pattern solutions that can be integrated with a tool: concerns that do not match the template might be better off with the un-refactored implementation.

8.3 RQ3: Separation of Concerns

Our case study had a satisfactory outcome in achieving a better separation and modularization of concerns in the targeted application. As we were able to notice, the crosscutting code is an important part of the refactored elements, in some cases, such as the *Command* elements, the crosscutting part constituted over 50% of the original code. The core code is easier to understand in the absence of the migrated crosscutting concerns because it isn't entangled with code that is not related to the core functionality. To understand the aspect code, on the other hand, one typically also needs to understand the base code that it advises. This is increased further by high coupling between the aspect and the base code, like for aspects that intercept calls from advised methods to reuse the values returned by such calls.

While refactored, crosscutting-free code is easier to comprehend, modifications to such code would still require awareness of the advice that applies to it. For instance, aspects might assume a certain order of the calls from an advised method, which has to be preserved to correctly introduce additional behavior.

Keeping track of the order of different advice in an aspect solution and preventing accidental changes might prove difficult, particularly when the number of aspects increases. The support from present development environments would not provide much insight into violations of such ordering, or into the ordering itself. This becomes more of an issue when the order is set using name-based wildcards, and new aspects match an existing rule for aspect precedence that should not apply to them. A similar situation might occur when changing an aspect solution that is already covered by a precedence rule, and the changes would not be compliant with that rule. Changing the position of an advice definition in an aspect could also modify precedence, if multiple advices in the aspect apply to the same joinpoints. Unspecified precedence could also lead to interference between new advices introduced by refactoring and existing ones [32]. Automatic refactoring needs to be aware of these issues.

Some concerns might be crosscutting for advices, similarly to the way they are crosscutting for methods. For instance, the re-use of specialized enumerations in JHOTDRAW requires to reset them after each iteration. Such enumerations are used by some advices in the aspect solutions. Applying aspect solutions to aspects might prove challenging for both tool support and comprehensibility.

8.4 RQ4/RQ6: Level of Automation

We showed that our approach supports a semi-automated implementation of all steps, including the final, refactoring one. The refactoring tool provides an open implementation for aspect code generation, which complements the incomplete support available in the AspectJ Development Tools (AJDT).

The tool support for refactoring, in particular, can be enhanced in a number ways, such as detection and ordering of multiple advice that apply at the same join point, or pointcut definitions that would allow us to capture a concern's intent rather than its extent. The latter feature requires that the pointcut definition in SAIR would be based on the query definition documenting a concern in SoQueT, rather than on the explicit set of results of the query that gives the crosscut elements.

The tool can also be improved to better deal with complex dependencies of the code to be extracted to aspects, such as conditional triggering of the advice.

However, some refactorings would unavoidably be difficult to execute automatically, such as the undo support concern in the *PasteCommand* example discussed in Section 5.4.2. In this case, due to a high degree of tangling, we needed to implement a helper-advice to capture the value of a local variable required both by the core functionality as well as by the crosscutting undo setup. This advice does not map onto a crosscutting concern, and so will not fit into any of our refactoring templates. This implies a manual solution, as well as a manual integration with the aspect solution of the undo crosscutting concern that uses the value captured by the helper-advice.

A particularly challenging automatic refactoring would be the one for *Redirection layer* instances: the original, dynamic solution uses a common interface for both redirectors and potential receivers. This interface hides the identity of the object for which a call is made. However, the refactoring of redirectors requires to know which calls are meant for a redirector and so need to be attached an advice introducing the functionality of the refactored redirector.

We believe that the case study presented in this paper is a required step before setting out to design semi-automated aspect refactoring tooling. The study gives us insight into the complexity of each refactoring and the trade-offs to be made. The challenges and limitations discussed in the previous sections also indicate that completely automated aspect refactoring is unfeasible in any practical situation, since the process requires a significant level of interaction with the users to guide the system through the right decisions.

An important benefit of the implemented automation consists of the challenge detection and resolution mechanisms, which allow the user to assess the impact of the refactoring, and assist with common solutions to the encountered problems.

8.5 RQ5: Differences between Manual and Semi-automatic Results

The tool-supported refactoring delivered to our expectations of enabling automatic transformation of the crosscutting Java implementation of concerns to AspectJ constructs. With this goal in mind, the results of the semi-automatic refactoring do not differ substantially from those of the manual approach.

Most differences appear in the design of the final aspect solution, where the manual approach provides more generic and flexible solutions. However, it is important to notice that most refactoring tools, ours included, aim at a basic solution that eliminates the tedious and error-prone work of code transformations. This is particularly relevant in the context of concerns scattered over a large number of modules. The tool's solution can then be improved through successive iterations, for example by abstracting reusbale aspects and pointcut definitions.

Major differences in results as covered in earlier sections include the defintion of pointcuts, where the current tool solution provides an enumeration of joinpoints, ordering of advice, or the current lack of support in the tool for nested type definitions in an aspect.

8.6 RQ6: Manual Effort with SAIR

SAIR uses a wizard-guided approach to refactoring, with each page of the multi-step refactoring process prompting the user for input to design the aspect-oriented solution. While some steps are straightforward to complete, such as selecting the aspect module to migrate the concern to, some other steps could be more challenging. In particular, the resolution step for steering the mapping of the concern to be refactored onto the template solutions provided by the tool requires a trade-off between the different solutions proposed by the tool.

However, most part of the manual work possibly needed for migrating to aspects would consist of code restructuring using object-oriented refactorings to allow for clear definitions of pointcuts and advice. This effort depends from case to case, on the level of tangling of the crosscutting concern to be refactored.

It is important to notice that the descriptions of concerns being passed to SAIR are created in SOQUET and this description directly impacts the complexity of the aspect-based solution to refactor the concern. The integrated approach to migration allows for a good matching between the sort queries in SOQUET and the template refactorings in SAIR; however, the goal of SOQUET to provide a comprehensive description of concerns can pose challenges to turning these descriptions (e.g., the set of call sites of a crosscutting method invocation) into a pointcut definition supported by AspectJ. Future work in improving the expressiveness of concern descriptions in SOQUET will not necessarily reduce the manual work required to restructure the code for refactoring.

8.7 Threats to Validity

A threat to external validity is that we studied only one system, JHOTDRAW, of moderate (20,000 lines of code) size. The choice for this case was primarily motivated by the common use of this system in other aspect mining and aspect refactoring research papers.

We furthermore conjecture that the size of the system is not the limiting factor. It may sometimes impose constraints on the aspect mining phase, which has to process the full code, but we have previously shown that our fan-in analysis method scales up to half a million lines of code [20]. For the subsequent steps, the size and spread of the concerns is what matters. Hence, it may be the case that the level of scattering could become a limiting factor: The more classes involved, the more complex the refactoring might be. Thus, in future work, cases in which a high level of scattering occurs would be of particular interest.

Related to this is the fact that JHOTDRAW has the particular characteristic that it was developed as a showcase for design pattern use. As a result it is very cleanly designed which may make it easier to recognize particular concerns and templates that form the basis for our migration. This may be different in other, less cleanly designed systems and needs to be investigated in future case studies.

Since our case study was explorative in nature and aimed at demonstrating the feasibility of semi-automated refactoring, our analysis is primarily qualitative, which can be considered a threat to construct validity. The identification of metrics for demonstrating improved separation of concern, however, is, to the best of our knowledge, an open research area. Nevertheless, in future studies more quantitative data would be highly desirable.

Last but not least, we attempted to mitigate threats to reliability by making as much material available on line as possible, including the concern models and the resulting refactored code.

9 Related work

While each step in the migration of crosscutting concerns has been addressed by related research, we are not aware of an integrated strategy like the one proposed in this paper.

The present approaches to aspect refactoring can generally be distinguished by their granularity. Laddad's set of refactorings cover both low level ones, such as *extract method calls into aspects* or *extract interface implementation*, as well as more complex refactorings, like design patterns, transactions management, or business rules [17]. Although the latter subset typically involves multiple concerns to be refactored, there is no categorization of these concerns or refactorings and they sometimes consist of nothing more than a name and motivation. This makes them less directly applicable to source code and requires more interpretation from the developer which cannot be automated.

Hannemann et al. propose an approach to the aspect refactoring of design patterns based on a library of abstract roles [13, 12]. The role-based refactoring requires one to map a pattern's implementation onto the predefined roles describing the pattern, and then applies a set of instructions to refactor the implementation to aspects. The approach is a step further towards generic, abstract solutions to typical problems that involve crosscutting functionality. However, as we have already seen, these patterns typically have a complex and variable structure in source code. Moreover, they generally consist of more than one atomic crosscutting concern. Refactoring a whole pattern in one step might prevent complete comprehension of the concerns involved. Moreover, our experience suggests that pattern implementations can vary significantly from a standard description and one-step refactoring could be hampered by complex dependencies. We cannot make a full assessment of this approach as the implementation and the experimental results are not available, but we believe that all the limitations discussed in this paper would equally apply to it.

Finer-grained refactorings have been proposed in the form of code transformations catalogs [11, 27] and AspectJ laws [6]. These transformations can occur as part of the aspect refactoring of an atomic crosscutting concern, but remain oblivious to the refactored concern. They describe the mechanics of migrating Java specific units to AspectJ ones (e.g., *Extract Fragment into Advice, Move Method/Field from Class* *to Inter-type*). Such small step transformations might benefit the implementation of automatic refactorings by preventing complex dependencies and ensuring behavior preservation as discussed by Cole and Borba [6]. However, more effort is required to assess their general applicability: for example, the case-study used for the refactoring in [27], is an *Observer* pattern implemented in a demonstrative application, which lacks the complexity of a real system like JHOTDRAW.

Finally, Tonella and Ceccato describe a list of refactorings based on the assumption that interfaces are often related to crosscutting concerns [34]. In [27] this is referred to as the *Interface Implementation* code smell and the authors provide refactorings to migrate part of the interface to an aspect in case it can indeed be regarded as a crosscutting concern. The authors of [34] describe a prototype tool for the refactoring of interfaces, which unfortunately is not publicly available. They have applied the tool to three Java packages from the discussion it follows that the concept works but some manual refinement of the resulting aspects was still required.

The idea of using program slicing for the extraction of crosscutting concerns was briefly discussed in [7]. In [8], Ettinger and Verbaere describe how program slicing can be used to extract aspects in the same way as is done for method or object extraction only to a new advice instead of a class. To make pointcut definition easier, standard object-oriented refactorings can be applied first. The autors provide no study on the general applicability of this approach, other than a simple example. The granularity is comparable to that of catalogue-based refactorings, i.e., acting on method level.

In comparison to the work on the fine-grained refactorings discussed above, the sort-based approach presented in this paper emphasizes concerns and identifies common properties at a consistent granularity level. This allows us to design a complete migration strategy, where the refactoring is integrated with steps for concern identification and comprehension.

Similar observations also apply to the comparison with the refactoring approach by Binkley et al. [1]. Their emphasis is on full automation, and they offer an Eclipse plugin for conducting six elementary refactorings. They focus on our fourth step only, and assume aspect mining has resulted in `@begin-aspect` and `@end-aspect` annotations in the code. As an example, one of their six refactorings moves individual calls to separate aspects, after which a non-trivial pointcut abstraction step is needed to merge the results. Our approach eliminates the need for this complex abstraction step thanks to the sort-based integration between aspect mining and refactoring, which allows our refactoring to be based on a full concern model. Like us, they use JHOTDRAW as one of their case studies. Somewhat surprisingly, they do not report any of the limitations that we identified, although their results exhibit the same limitations.

A refactoring approach for migrating crosscutting concerns in C code to domain-specific aspects is described by Bruntink et al [4]. Their approach involves the recognition of specific concerns implemented by means of C code idioms, the design of a domain-specific aspect language to represent them, and the automatic migration of idioms to this new language. They apply their approach to an idiom used to conduct null checks on parameters for C functions. An in-depth account of the difficulty of migrating the return code idiom as used in C to mimic exception handling towards a more declarative, aspect-based solution is described in a later paper by Bruntink et al [3].

10 Conclusion

In this paper, we proposed an integrated strategy for migrating crosscutting concerns to aspect-oriented programming. We presented in detail the refactoring step of our strategy, and applied the entire migration process to concerns in an open-source application. Furthermore, we discussed the challenges of refactoring crosscutting concerns to aspects and how these could impact the design and implementation of automatic aspect refactoring.

The contributions of this work can be summarized as:

- An integrated strategy for migrating crosscutting concerns to AOP solutions;
- An aspect refactoring approach based on crosscutting concern sorts and a set of refactoring templates and the translation of this approach into a method to semi-automatically perform aspect-introducing refactorings based on crosscutting concern sorts;
- A report on our experience with manually migrating concerns in a real system to aspects and the challenges of this process. This report is useful for both assessing existing support for aspect refactoring as well as for determining how one can automatically refactor various categories (that is, sorts) of crosscutting concerns;
- AJHotDraw, a manually created show-case for aspect refactoring in an open-source implementation that can be further used by researchers and practitioners to evaluate aspect-based solutions to crosscutting concerns;
- Details of aspect refactoring algorithms for two crosscutting concern sorts which are implemented in our semi-automatic refactoring tool SAIR. The tool also implements a resolution mechanism that assists the user in dealing with the challenges that raise from complexity of refactoring by providing a set of predefined solutions to common challenges.
- A report on our exploratory case study in which we applied SAIR on JHotDraw and compared its results with the manually migrated AJHotDraw.

Opportunities for future work AJHotDraw provides a code base for related research to measure code improvements due to aspect code. Furthermore, this work provides us with the hands-on experience for designing and implementing sort-based aspect refactoring. We plan to integrate the refactoring tool SAIR with SoQueT, our tool for concern documentation. The refactorings would apply to each query documenting a sort instance, and hence benefit from the description of the concerns available by the query results. We aim to implement refactoring support for all the sorts available in SoQueT. Additional work will be carried out to identify additional challenges that could occur in automatic refactoring to aspects, and to include solutions to these potential challenges in the tool.

References

1. D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.
2. Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)*, pages 221–230. IEEE Computer Society, 2006.
3. M. Bruntink. Reengineering idiomatic exception handling in legacy C code. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, pages 133–142. IEEE Computer Society Press, April 2008.
4. M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pages 37–46. IEEE Computer Society, 2005.
5. M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, 2006.
6. L. Cole and P. Borba. Deriving refactorings for AspectJ. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 123–134. ACM, 2005.
7. A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In L. Thavildari and K. Kontogiannis, editors, *Proc. WCRE Workshop on REFactoring: Achievements, Challenges, Effects*, Waterloo, Canada, 2003. University of Waterloo.
8. R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development (AOSD)*, pages 93–101. ACM Press, 2004.
9. R.E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.
10. E. Hajiyev, M. Verbaere, and Oege de Moor. Codequest: Scalable source code queries with datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 2–27, 2006.
11. S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In *Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World(Net.ObjectDays)*, pages 19–35. Springer-Verlag, 2003.
12. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of 17th Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pages 161–173. ACM Press, 2002.
13. J. Hannemann, G.C. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 135–146. ACM Press, 2005.
14. William Harrison, Harold Ossher, Stanley Sutton, and Peri Tarr. Concern modeling in the concern manipulation environment. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM.
15. D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187. ACM Press, 2003.
16. Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect Oriented Software Development*, 4640:143–162, 2007.
17. R. Laddad. *AspectJ in Action - Practical Aspect Oriented Programming*. Manning Publications Co., 2003.
18. N. Lesiecki. Aop@work: Enhance design patterns with AspectJ. `www-128.ibm.com/developerworks`, May 2005.
19. M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. PhD thesis, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, January 2008.
20. M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.
21. M. Marin, L.Moonen, and A. van Deursen. A classification of crosscutting concerns. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 673–677. IEEE Computer Society, 2005.

22. M. Marin, L. Moonen, and A. van Deursen. A common framework for aspect mining based on crosscutting concern sorts. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 29–38. IEEE Computer Society, 2006.
23. M. Marin, L. Moonen, and A. van Deursen. Documenting typical crosscutting concerns. In M. Di Penta and J. I. Maletic, editors, *Proceedings 14th Working Conference on Reverse Engineering (WCRE)*, pages 31–40. IEEE Computer Society, 2007.
24. M. Marin, L. Moonen, and A. van Deursen. An integrated crosscutting concern migration strategy and its application to JHotDraw. In B. Korel and M. W. Godfrey, editors, *Proceedings of the 7th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 101–110. IEEE Computer Society, 2007.
25. M. Marin, L. Moonen, and A. van Deursen. SoQueT: Query-based documentation of crosscutting concerns. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007.
26. Kim Mens, Bernard Poll, and Sebastián González. Using intentional source-code views to aid software maintenance. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 169–178, Washington, DC, USA, 2003. IEEE Computer Society.
27. M.P. Monteiro and J.M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122. ACM Press, 2005.
28. R. van der Rijst. Sort-based refactoring of crosscutting concerns to aspects. Master's thesis, Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, 2008.
29. Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3, 2007.
30. M.P. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416. ACM Press, 2002.
31. David Shepherd, Jeffrey Palm, Lori Pollock, and Mark Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *Proceedings of the 20th International Conference on Automated software engineering (ASE)*, pages 184–193, New York, NY, USA, 2005. ACM.
32. M. Storzer and F. Forster. Detecting precedence-related advice interference. In *Proceedings of the 21st International Conference on Automated Software Engineering (ASE)*, pages 317–322. IEEE Computer Society, 2006.
33. P. Tarr, W. Harrison, and H. Ossher. Pervasive query support in the concern manipulation environment. Technical Report RC23343, IBM TJ Watson Research Center, Yorktown Heights, NY, 2004.
34. Paolo Tonella and Mariano Ceccato. Migrating interface implementation to aspects. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pages 220–229. IEEE Computer Society, 2004.