Testing common refactorings of crosscutting concerns in Java to AspectJ

Master's Thesis

Gijs Peek

April 9, 2008

Testing common refactorings of crosscutting concerns in Java to AspectJ

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Gijs Peek
born in Woudrichem, the Netherlands

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

Cover picture: another detail from one of the wallside mosaics in the mausoleum of Galla Placidia (386-452) in Ravenna, Italy.

Testing common refactorings of crosscutting concerns in Java to AspectJ

Author: Gijs Peek
Student id: 9644357
Email: gijs.peek@gmail.com

Abstract

Aspect-orientation is a programming paradigm that provides a novel way of moduralizing *crosscutting concerns* - concerns in software that would otherwise get scattered among and/or tangled with the core program.

While a number of common refactorings exist for migrating these crosscutting concerns in existing implementations to aspect-oriented solutions, verifying the correctness of these refactorings remains little more than guesswork. This thesis describes how to solve this problem by *creating automated tests for verifying common refactorings of crosscutting concerns in Java to AspectJ*.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Ir. M. Marin and Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Committee Member:	Drs. P.R. van Nieuwenhuizen, Faculty EEMCS, TU Delft
Committee Member:	Dr. Leon Moonen, Faculty EEMCS, TU Delft

Preface

First and foremost I would like to thank my first and second supervisors, Arie and Marius, for their guidance, support and reviewing of my work. I would also like to thank Leon Moonen and Peter van Nieuwenhuizen for reviewing this thesis.

Robin and Joost deserve some credit if only for their patience while listening to my usual rantings about whichever subject I was working on during my thesis. My thanks also goes out to my parents for their continuing support, without which it would have been impossible for me to get to the point where I currently am. I would also like to thank my wife, Hilde, for everything she has done to support me. And finally, to everyone I have failed to mention above, thank you.

Gijs Peek
Delft, the Netherlands
April 1, 2008

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

> "Using a computer program is much like parachute jumping: if you fail the first time, you're not likely to try again" – anonymous

Delivering a quality product is an important asset in a line of business where your customers can easily trade your product for your competitor's. This is one of the reasons why the software industry is constantly looking for new means of creating better software within a project's time and budget constraints. This thesis investigates the combination of two such techniques: aspect-oriented software development and automated software testing, both of which we will now briefly introduce.

Aspect-oriented software development (AOSD) is a relatively new programming paradigm (The term was coined by Kiczales et al. [23] in 1997). It has evolved from technologies which addressed the issue that for some programming problems, as they state, "neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement". The programming problems they are referring to are the ones that crosscut the program's basic functionality, for which the implementation with both procedural and object-oriented programming techniques gets scattered and tangled throughout the code. These are called *crosscutting concerns (CCCs)*.

AOSD provides a way of modularizing these cross-cutting concerns much like object-oriented programming enables modularizing common concerns. However, these new code localization and modularization techniques do pose new challenges for testing.

Software testing is a long-standing practice which aims to find defects in software. It can already be applied during development to detect, and in this way enables us to repair these as early as possible. This can be beneficial to the project, as McConnell tells us "projects that remove about 95 percent of their defects prior to release are the most productive; they spend the least time fixing their own defects" [33]. As evaluated later on in chapter 3 on page 19, only limited results have yet been achieved in finding methods for testing aspect-oriented programs.

1.2 Problem statement

The goal of this report is to *find out how to test common refactorings of crosscutting concerns in Java to AspectJ*.

As we have not encountered the the terms AspectJ and refactorings before, we will briefly explain them here:

***AspectJ* [22]** is an aspect-oriented extension to the Java programming language. We have chosen to focus on this language since it currently enjoys the most widespread support.

***Refactorings* [17]** are internal design improvements in a program's code that do not change its behavior. They are usually applied as a defense against the dreaded *software erosion* effect. This is the gradual deterioration of a program's design. Software maintenance and implementation of new features are typical causes of this deterioration.

Our project is based on M. Marin's PhD research, in which he has classified a number common refactorings [31], and performed a number of these refactorings in a case study [28, 13, 30, 27]. This case study is the AJHotDraw(AJHD)[1] project, an aspect-oriented refactoring of the JHotDraw(JHD)[2] project. The JHD project is commonly used as an aspect mining case study [10, 38, 40, 48]. It implements a two-dimensional graphics framework, and contains a small number of similar draw applications. See figure 1.1 on the facing page for an example screen of one of these applications.

AJHD is an exploratory case study, contrived for exploring possible refactorings of object-oriented code to aspect-oriented code. As JHD is geared specifically towards programming with patterns and good programming practice, its refactoring potentially provides a good example for demonstrating the added value of using aspects for programming. The AJHD project already contained a number of aspect-oriented refactorings before we started out. We will show how we have managed to test these, and fill the gaps where they have not been applied yet.

Our search for finding viable test methods was also of an exploratory nature. We did not attempt to come up with an exhaustive list of testing techniques (this goal might arguably not even be achievable), but we focused on finding faults that are specific to aspect-oriented programs, and most importantly to find some testing techniques that can address these faults. We consider this project a success *if we can successfully test the refactorings that were applied to JHD*, successfully in this case meaning we are under the impression that all conceivable likely faults have been covered in the test cases.

[1] `http://sourceforge.net/projects/ajhotdraw/`
[2] `http://sourceforge.net/projects/jhotdraw/`

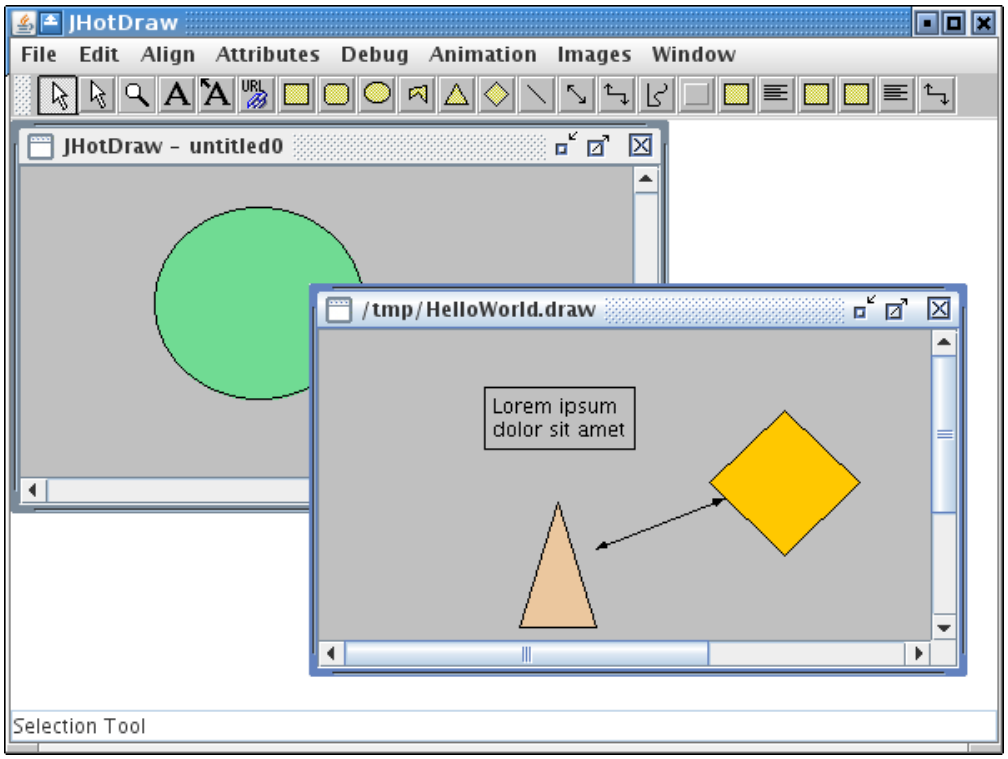

Figure 1.1: The JavaDraw application from the JHotDraw project

1.3 Chapter Structure

We will start out by more detailed explanations of the background concepts needed to understand the rest of the thesis in chapter 2 on page 5. We will then proceed by outlining the work that has already been done in this field of research in chapter 3 on page 19. This work will be the baseline for the rest of the thesis. When this has been established, we will outline our ideas for testing aspect-oriented programs in chapter 4 on page 25, and subsequently apply them to the AJHD case study in chapter 5 on page 43. Then in chapter 6 on page 65 we will present a special and most complicated case in aspect-oriented programming, namely the implementation of contract enforcement mechanisms. Finally we will discuss the results in chapter 7 on page 75 and outline our conclusions in chapter 8 on page 81.

Chapter 2

Background

Before we actually start testing the various aspect-oriented constructions, we will go over some essential background information. This chapter explains the concepts that have been introduced in chapter 1 on page 1 in more detail. We will first take a look at the refactoring process, with which AJHD is mainly concerned. Then we will move more to providing a basis on the thesis's testing part by looking at how object-oriented systems are tested. Then we will introduce aspect-oriented development, after which we will skim over the related research on testing aspect-oriented systems. We will finish by outlining the AJHD case study on which we will be applying new testing techniques further on in this report.

2.1 Refactoring

2.1.1 About refactoring

Refactoring improves the design of existing code, which reduces future programming effort. This technique is applied in the AJHotdraw project[1], which is an aspect-oriented refactoring of the JHotDraw project[2], version 6.0b1. This section briefly describes and motivates the concept of refactoring. M. Fowler et al. describe two meanings of the word refactoring, of which ate meaning is based on its context: [17]

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

If we dissect the first definition we find that:

- A refactoring changes the internal structure of a program: Refactoring means tracking down some part of the code that we do not like, and restructuring it for the better.

[1] `http://sourceforge.net/projects/ajhotdraw/`
[2] `http://sourceforge.net/projects/ajhotdraw/`

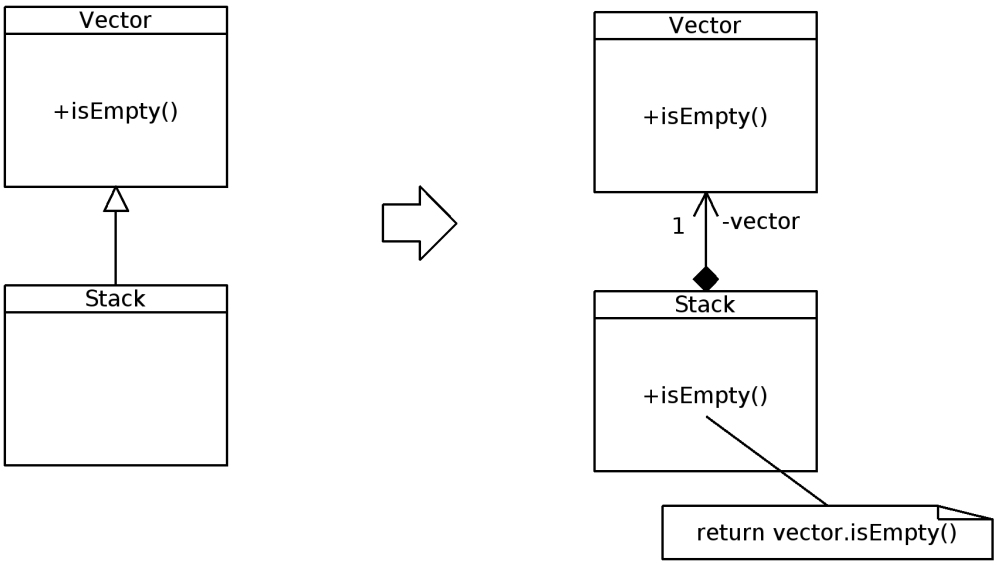

Figure 2.1: Inheritance to delegation refactoring

- A refactoring makes the code easier to understand: Good programmers know that the first attempt at writing something does not usually produce a very clean, easy to understand implementation. Refactoring afterwards, if applied correctly, improves code understandability.
- A refactoring makes the code cheaper to modify: Computer programs are not static entities, they are in constant need of maintenance, whether by changes in requirements or detection of bugs. A side-effect of performing this maintenance is software erosion: its structure slowly degrades as small changes are applied, increasing the maintenance cost. Refactoring is a way to revert this erosion, and keep the maintenance effort within bounds in the long run.
- A refactoring does not change the observable behavior of the software: Refactorings make changes in the software structure, not its function. End users and other programmers working on different parts of the system should not be able to notice the difference.

Figure 2.1 gives an example of one of the many refactorings Fowler et al. describe. Here, an inheritance relation is replaced with delegation, which can be used if a subclass uses only part of the superclass interface or should not inherit its data.

2.1.2 Testing Refactorings

When we are refactoring code, it is a good idea to check whether we are not breaking something while doing it. Ideally, the code structure has improved while its observable behavior has not. M. Fowler et al. propose solving this by means of applying a solid set of tests to the code and applying it both before and after a refactoring [17]. While these tests won't be able to discern improvements in the code structure, they *can* indicate something about changes in its observable behavior however: if all tests succeeded

both before and after the refactoring, this indicates that both versions provides the same observable behavior. Note, however, that this is only an indication. Testing alone cannot actually prove this.

As A. van Deursen et al. noted [14], it may not be possible to use the same, unmodified test set on the system after a refactoring as was used on the system before the refactoring. Refactorings can change the interface the tests expect (e.g., when a method is renamed or moved to another class), rendering them incompatible with the core system. Van Deursen et al. propose to solve this by providing the testcases with a backwards compatible interface if possible (and thus not having to make any changes to the test classes), and only to change the test classes if making such an interface is not possible. Also, additional test cases may be required afterwards for the following reasons:

- the refactoring may have extended the interface or added new structures, and
- the act of refactoring itself may introduce new bug hazards (see also [43]).

Some tests may also be completely invalidated by a refactoring: take for example the inheritance to delegation refactoring we used earlier (illustrated in Figure 2.1 on the facing page). The test suite for this system might contain tests that check whether the state of objects from the Stack class is not invalidated by calling methods it inherits from its Vector superclass, like the `inserElementAt()` method. These tests will not be necessary for the refactored system any longer, as the methods they tested simply cannot be invoked on Stack objects anymore.

2.2 Testing Object-Oriented Systems

An extensive overview of the techniques for testing object-oriented systems is provided by Binder [5]. Although the object-oriented test patterns that Binder describes may not yet be sufficient to test aspect-oriented projects, his testing approach can serve as a starting point for devising testing approaches that are. We will focus on those parts of Binder's rather sizable work which we will be able to use in our subsequent search for testing approaches for aspect-oriented programs.

We will go over the process of creating good test strategies; *why* you should test your code, *how* to test it and *what* elements to test. We will also look at bug hazards in object-oriented code as some of these are similar to bug hazards in aspect-oriented code, and in this case a similar testing strategy for this hazard type could probably be applied.

2.2.1 Why test object-oriented code

Why should we test our code?
Testing is necessary to produce high-quality, reliable, safe and successful object-oriented applications [5] while being able to save over 50% of the development effort [36]. Object-oriented software is no less bug-prone since it is generated by the same gray matter that produces any other kind of code. In fact, testing is accepted by leading practitioners as a necessary part of object-oriented development [16, 20, 8, 21]. Testing

can reveal bugs that would be too costly or impossible to find with other verification and validation techniques.

Object-oriented software poses the next challenge to testing. While methods like black boxes apply to object-oriented software as well, mechanisms like inheritance, polymorphism, late binding and encapsulation cannot be compared. These mechanisms require new testing approaches to be tested effectively. Furthermore, since object-oriented development is iterative and incremental, test planning, design and execution must be similarly iterative and incremental.

Summary We should test our code because it provides opportunities to make better object-oriented programs with less development effort.

2.2.2 How to test

How should we test our code?
Software testing is the execution of code using combinations of selected input and state to reveal bugs. Since testing the entire search space (all input/state/execution sequence combinations) is computationally infeasible even for simpler programs, we can only test a subset of the possible combinations, and we should make a test selection. We can make this selection based on a *fault model*, which allows us to make an educated guess as to which combinations warrent our effort in adding them to our test suite.

Any rational testing strategy must be guided by a *fault model* which provides relationships and components that are most likely to contain faults. There are two widely used general fault models and corresponding test strategies: *conformance-driven testing* (also called black-box or behavioral testing), which validates conformance to requirements and specifications, and *fault-driven testing* (also called white-box or implementation-based testing), which seeks to reveal implementation faults. The first one relies on a nonspecific fault model since any fault suffices to prevent conformance, while the second one requires a specific fault model to direct the search for faults, preferring more likely faults over other, less likely ones.

The testing process should be primarily conformance-driven since conformance-driven techniques tests what it *should* do (i.e., conforms to its specification), while implementation-driven testing techniques can only show that the code does what it does and not necessarily what it should do. Implementation-driven techniques can be used as a test *adequacy criterion* (explained later in section 2.2.3 on page 11) to measure the adequacy of the test suite (e.g., by applying code coverage analysis) and to guide the testing process: the implementation can be used as a source of information on program structure and design.

Summary We should build our test suite conformance-driven, based on a fault model because this is the most effective way to find faults: its mainstream alternative, implementation-driven testing, should not drive the testing process since it can only show that the code does what it does, while conformance-driven techniques test what it *should* do. A fault model is needed to make a rational selection of tests from the space of all possible tests.

2.2.3 What to test

What should we test?
This is probably the most difficult question to answer, and it may take some experience to be able to determine for a given system which test strategies should be applied to it in order to test it effectively. Using test strategies that have been expressed in the form of *test design patterns* can ease this decision since a test design pattern (as described by Binder), apart from the testing approach and some additional information, provides directions about its applicability and consequences.

We can select which of the test patterns we know about can be applied to the system we are testing. Additionally, if we can determine the most likely faults in our system, we can try to find test strategies that cover those faults. We usually look at the system design to determine which faults are most likely. For instance, if we have captured our system requirements as use cases, we can apply the "extended use case test design pattern" [5, p. 722-731] to capture the most likely inconsistencies between the implementation and those use cases. However, the resulting test suite will probably not capture a lot of faults that result from errors in conditional logic or forgetting an isEqual() implementation. To select the correct test patterns, we should be aware of the specific bug hazards which our code exhibits. We should also be aware of the fact that testing should be applied at all system scopes. We will explain both these points in more detail below:

Bug hazards in object-oriented code

Object-oriented code is subject to new bug hazards in relation to procedural code. A general overview of these hazards is included below:

- *Encapsulation* is the mechanism that controls the visibility of names in and among lexical units. They do not directly contribute to the occurrence of bugs, but can hamper testing efforts by denying access to testable system features.

- *Inheritance* supports reusability by allowing shared and different entities of a common entity to be represented, and supports definitions of type and subtypes, allowing for efficient extensibility [35]. It can be used as a model of hierarchy, a control mechanism for dynamic binding and as macro substitution for programmer convenience.
  Inheritance weakens encapsulation, and can be used in many erroneous ways. Bug hazards include:

  - Incorrect initialization or forgotten methods: forgetting super.init() call, forgetting to override a specific method, like isEqual(). Methods may also be reused accidentally this way.
  - Mixing problem domain relations and shared implementation features in a class hierarchy, or representing an incompatible subtype relation. These may cause irregular and complex type hierarchies.
  - Failing to adhere to the Liskov substitution principle [26] or Meyer's design-by-contract approach [34].

As aspect-orientation provides new constructs for code reuse, testing techniques that focus on inheritance-related concepts might be able to be adapted for them.

- *Polymorphism* binds a reference to different classes of objects. This reference is declared in a server object, and the receiver (client) object is determined at runtime.
  Bug hazards created by polymorphism include:
  - Changes to the server object may affect client objects.
  - Minor coding errors may bind messages to the wrong server.
  - Polymorphic message code is deceptively simple, while the actual behavior often is not.
- Method behavior often depends on and/or influences object *state*. As a result, messages must be sent in some order, raising an issue: which sequences are correct, and which are not?

Testing scope

If we test our system, we typically start by testing the smallest individual modules (i.e., methods). If we have done that to our satisfaction and maybe have run the application once or twice without detecting any faults, we might be inclined to think we have adequately tested our system. However, Weyuker's testing axioms for modular systems show us that this is not the case [45].

- The *antiextensionality* axiom states that a test suite that covers one implementation of a specification does not necessarily cover a different implementation of the same specification.
- The *antidecomposition* axiom states that the coverage achieved for a module under test is not necessarily achieved for modules that it calls.
- The *anticomposition* axiom states that test suites that are individually adequate for segments within a module are not necessarily adequate for the module as a whole.

These axioms can been interpreted specifically for subclass testing [37], but also indicate that all program structure levels (method/combinatorial, class, subsystem and application level) can have their own bug hazards, the ones at higher levels created by the extension of and interactions between lower-level units. Binder also describes test patterns for various structure levels as well as some model-specific ones like the state machine test design strategy. He also reminds us to design these testcases scope-upwards, so that faults that should be tested by lower-level tests do surface there and not a the higher level test suite, because then they are much harder to trace down to their source.

Test adequacy

So how do we know when to stop looking for more test cases? We cannot simply state that we want to test 'adequately' or 'thouroughly' without being able to quantify the 'thoroughness' in some manner. This is what test adequacy encompasses. An *Adequacy criterion* specifies the elements of an Implementation Under Test (IUT) to be exercised by a test strategy [44]. We can then measure the *coverage* of a test suite by measuring the the amount of these elements that have been exercised. We have listed a number of notable adequacy criteria:

- Statement coverage: All statements in a method must be executed. Also known as line coverage, since usually a line contains at most one statement.
- Branch coverage: Each branch must be executed. This implies that each predicate expression outcome must be exercised at least once.
- Multiple-condition coverage: Each combination of boolean values in each compound statement must be executed.
- Method coverage: Each method must be executed.
- Object code coverage: Measures coverage over the compiled (object) code. The usage of this criterion is discouraged as it is more difficult to interpret than source code coverage and has not proven to exhibit any advantages over source code coverage.
- Basis-path model: Based on the cyclomatic complexity [32] of the code. It requires at least C (C being the cyclomatic complexity) testcases with different entry-exit paths. Using this criterion is also discouraged, because it is unreliable (statement nor branch coverage is guaranteed).
- Data Flow coverage: This criterion focuses on actions that are performed on variables. It describes coverage of execution paths starting with the definition of a variable and ending at its usage or destruction. Numerous variations on this criterion have been suggested.

Using one or several adequacy criteria, we can calculate the amount of our code that was covered, and more interestingly, the code that was not covered. This can point out several mistakes, including:

- possible faults that were not or inadequately exercised by our test suite
- surprises in the code
- badly designed code constructs ('if it's hard to test, it's probably not well-designed')

Good test coverage can provide confidence in both the completeness of the test suite and the correctness of the program. However, we should always be aware of the fact that reaching a certain coverage level however only shows that the covered code has been executed during the test, not how well its results have been checked: this is

something we have to determine ourselves. Complete code coverage also does not yet mean all possible faults have been covered: the coverage criteria still describe a subset of all faults which we deem more likely to appear than others. Testing never shows the absence of bugs: it can only detect its presence.

Summary To know what to test, we should be aware of the possible faults our system could exhibit. We can then apply specific test patterns that exercise these faults. Coverage analysis can be of help in determining which faults we have missed, but it only provides limited information about the faults we have hit.

2.2.4 Application to Aspect-Oriented Systems

Aspect-orientation provides some new language constructs, as will be explained in the next chapter. These naturally come with their own fault hazards. Essentially, the testing problem remains the same: we should choose a small number of tests out of a large number of possible ones. The same general approach can thus be applied: we should apply conformance-driven tests, based on a fault model. We will also need new testing strategies to cover the newly found fault hazards.

2.3 Aspect-Oriented Software Development

> *This is what I mean by "focusing ones attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from one aspect's point of view, the other is irrelevant. [15]*

In order to be able to test aspects, we must first understand the anatomy of the aspects themselves. This chapter introduces aspect-oriented software development (AOSD) along with some common terms and considerations that are associated with it. AOSD provides techniques that, among other things, can improve the separation of concerns over the more traditional object-oriented approach. It is especially well-equipped to address *crosscutting concerns*, explained below.

2.3.1 Crosscutting concerns

A concern is a very broad concept which refers to any particular piece of interest or focus of a program. A concern could for example be a feature ("Help is available for all buttons"), specification verification (assertions, design contracts) or a constraint ("the application must respond within 1 second when button X is pressed"). Concerns should ideally be considered separately while developing a program [15] in order to decrease the complexity of the development. Techniques like modularity and encapsulation can be used to separate concerns. A good separation of concerns is achieved when each concern is implemented in one or more modules that are dedicated to that concern, which have few connections with the modules dedicated to the other concerns while being highly cohesive themselves.

A crosscutting concern is a concern which affects (crosscuts) other concerns. In object-oriented systems these concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either *scattering*

```
1  import java.util.Observable;
2  public class ObservablePerson extends Observable {
3      private String name;
4      private int age;
5      private Sex sex;
6
7      public void setName(String newName) {
8          name = newName;
9          setChanged();
10         notifyObservers(name);
11     }
12
13     public void setAge(int newAge) {
14         age = newAge;
15         setChanged();
16         notifyObservers(age);
17     }
18
19     public void setSex(Sex newSex) {
20         sex = newSex;
21         setChanged();
22         notifyObservers(sex);
23     }
24
25     public enum Sex { Male, Female };
26 }
```

Listing 2.1: An observable person, the object-oriented solution

(Section 2.3.1) or *tangling* (Section 2.3.1 on the next page) of the program, or both. Scattering and tangling usually occur together.

Scattering

Scattered code is code that belongs to one concern, but is distributed throughout many programming modules. An example of scattered code is the implementation of the Observer pattern [18], where in each subclass of Observable the code for notifying observers is scattered among its methods. Each time the state of an Observable has changed, it needs to notify its observers through the notifyObservers() method, so calls to the notifyObservers() method all methods from the that change the state of the Observable classes, instead of all being defined in one single spot. This is shown in Listing 2.1. The notifyObservers() calls are present in all three setter methods, 'scattered' throughout the class implementation.

You might notice that each notifyObservers() call is preceded by a call to the setChanged() method. This method sets the Observable changed state. Unless this is set, notifyObservers() will not actually notify any observers, and the notifyObservers() call will reset the changed state, so each notifyObservers() call should be preceded with a setChanged() call in order to notify the observers.

Tangling

Tangled code belongs to different concerns, but it is contained in one programming module. The Observer pattern also contains tangled code: every method which changes the state of an Observable class contains code for at least 2 concerns: its core functionality, which does the actual work the method is written for, and notifying the Observer objects that something has changed using the setChanged() and notifyObservers() calls. The ObservablePerson class in listing 2.1 on the previous page also shows code tangling, since each setter method performs two functions: setting the value it was intended to do, which is its core functionality, and updating the observers, which belongs to the observable concern. The code for two concerns are tangled in each method.

2.3.2 What are aspects?

Aspects have been introduced to address crosscutting concerns by modularizing them. Extensions which add aspects are available for many programming languages, but this paper focuses on AspectJ for the Java language since this is the most widely supported language around at the time of this writing. Aspect-capable compilers like ajc[3] or abc[4] enable aspects to be used as class-like language elements.

An aspect increases modularization by applying a join point model. Such a model consists of three elements:

- locations where code can be applied to (*join points* in AspectJ)
- a way to aggregate these locations (*pointcuts* in AspectJ)
- some method of applying code to them (*advice* in AspectJ)

These elements allow a single piece of code to be applied in many locations throughout the base program, without having to change pieces of code throughout the program. This way the implementation of a crosscutting concern can be limited to one or few program modules, while still being able to affect many code locations. This can make the codebase easier to understand and maintain as it improves the separation of concerns.

2.3.3 Aspects in AspectJ

A more detailed description of the AspectJ language constructs is included below.

join point: A join point is a point of execution in a (Java) program. Join points are predefined by the aspect-capable compiler, and include among others method call and execution, exceptions, class initialization and parameter changes.

pointcut: A pointcut describes a collection of join points. Pointcuts are used to separate function (what) from location (where), and provide an easy method to select a number of pointcuts that is similar to creating a search query for a database or creating a regular expression to match certain text parts.

[3] `http://www.eclipse.org/aspectj/`
[4] `http://abc.comlab.ox.ac.uk/`

```
1  import java.util.Observable;
2  public class ObservablePerson extends Observable {
3      private String name;
4      private int age;
5      private Sex sex;
6
7      public void setName(String newName) {
8          name = newName;
9      }
10
11     public void setAge(int newAge) {
12         age = newAge;
13     }
14
15     public void setSex(Sex newSex) {
16         sex = newSex;
17     }
18
19     public enum Sex { Male, Female };
20 }
```

Listing 2.2: An observable person, the aspect-oriented solution - base class

```
1  public aspect ObservablePersonAspect {
2      pointcut PersonFieldChanged(ObservablePerson p, Object value):
3          set(* ObservablePerson.*)
4          && this(p) && args(value);
5
6      after(ObservablePerson p, Object value):
7          PersonFieldChanged(p, value) {
8          p.stateChanged(value);
9      }
10
11     private void ObservablePerson.stateChanged(Object o) {
12         setChanged();
13         notifyObservers(o);
14     }
15 }
```

Listing 2.3: An observable person, the aspect-oriented solution - aspect

advice: Advice is a piece of executable code which is applied to a number of join points. Advice is accompanied by a pointcut which describes which join points to apply the advice to.

Additionally, an aspect may insert fields or methods in the object model. This is called an *intertype declaration*:

intertype declaration: An intertype declaration is a declaration that cuts across classes and their hierarchies. An intertype declaration may declare a field, a method or an inheritance relation outside the parent type (in an aspect).

Listings 2.2 and 2.3 contains an example of an AspectJ aspect: here, the cross-cutting code from the observable concern is separated from the core code by the Ob-

servablePersonAspect. In this aspect, the setChanged() and notifyObservers() calls are executed by the aspect each time a field value in an ObservablePerson has changed. The pointcut PersonFieldChanged selects all join points where a field belonging to an ObservablePerson is set to a new value. The after advice that is executed when this value has been updated calls the stateChanged() method that has been added to the ObservablePerson class with an intertype declaration. In the stateChanged() method, the setChanged() and notifyObservers() methods are called.

Inheritance

AspectJ defines its own inheritance mechanism for aspects. This mechanism is similar to inheritance in Java, with a number of notable differences. We have summarized the AspectJ inheritance rules below:

- An aspect (concrete or abstract) may extend a class or an abstract aspect. Concrete aspects may not be extended.
- Classes may not extend aspects
- An aspect may implement any number of interfaces
- An abstract aspect may define abstract and concrete pointcuts, concrete aspects may only define concrete pointcuts
- A concrete aspect must implement all abstract and interface methods and abstract pointcuts
- Concrete pointcuts may be overridden by subaspects (access modifiers apply as in Java, e.g., private pointcut cannot be accessed or overridden in subclasses).

2.4 The AJHotdraw case study

JHotDraw (`http://sourceforge.net/projects/jhotdraw/`) is a two-dimensional graphics framework which started out as a case study for an applied design patterns seminar, after which a number of enthusiastic software developers continued maintaining and expanding the project.
AJHotdraw (`http://sourceforge.net/projects/ajhotdraw/`) is an aspect-oriented refactoring of the JHotDraw project (version 6.0b1), which contains two modules: one for the refactored source (AJHotDraw), and another one (TestJHotDraw) that contains the test cases which can be run on both the original and refactored programs. TestJHotDraw is developed as a test subproject aimed at ensuring behavior conservation between the two solutions. We are going to use the TestJHotdraw project as a showcase for verifying behavioral equivalence between JHotdraw and AJHotdraw.

With behavioral equivalence we mean, in plain English, that it 'does the same thing'. In the most strict sense, this will probably never be entirely true for refactored code, since refactoring changes the design of the code, and since the program has changed is does not behave exactly the way it did before. If you stop looking under the hood, though, this property can be maintained. When for instance the code body of a single method has changed, equivalence would mean that the two method versions

would produce the same result for the same input. If the method would have a contract, this constraint could be relaxed to the point where the two method versions should only produce the same result when offered input that adheres to its preconditions, since only then does it guarantee its postconditions to be true. If one or more preconditions are not met, no guarantee can be made about the postconditions. A. van Deursen et al. have already defined a general testing approach [43] and applied them to some refactorings [28], which they identified for common types of crosscutting concerns [31, 30, 27].

Chapter 3

Related research

Some research on testing aspects has already been performed, but this work is mostly preliminary and formal testing strategies for aspect-oriented systems are still in their infancy. This section contains an overview of the research, which can be used as a basis to create test strategies for aspect-oriented systems.

As explained in Section 2.2.4 on page 12, aspect-oriented code should be tested in the same manner as object-oriented code. Firstly we will outline the fault models that have been developed, and look at some aspect-oriented adequacy criteria. Then we take a look at the testing strategies that are currently available for aspect-oriented systems. Finally we will go over some miscellaneous research that can be used in forming new testing strategies for aspect-oriented systems.

3.1 Fault models

Alexander et al. describe a fault model for testing aspect-oriented programs [1]. They state that testing aspect-oriented software is hard because aspects do not have an independent entity or existence, aspect implementations can be tightly coupled to their woven context, control and data dependencies are not readily apparent from the source of aspects or classes, and because of behavior that emerges from the weaving process.

They make a distinction between 4 types of faults:

1. faults in the core system
2. faults in the aspects isolated from the woven context
3. faults that emerge by interactions between one aspect and the core system
4. faults that involve multiple aspects

Whenever aspects are involved, the testing effort is likely to be much greater.

They also enumerate a number of potential faults in AOPs:

- Incorrect pointcut strength: pointcut patterns that are too strong or too weak
- Incorrect aspect precedence: introduced advice blocks may have to be executed in a certain order to function correctly

- Broken postconditions: invasive advice may compromise behavioral contracts by violating postconditions
- Unsatisfied state invariants
- Incorrect focus of control flow: join points that should only be selected in a particular execution context could result in failures which are difficult to diagnose
- Incorrect changes in control dependencies

Ceccato et al [11] later extended this fault model with three new faults:

- Incorrect changes in exceptional control flow (e.g., advice throwing an exception, or by using the `declare soft` statement)
- Failures due to inter-type declarations: these failures are similar to the ones due to normal declarations
- Incorrect changes in polymorphic calls: bug hazards caused by incorrect changes in method overrides

The final two bug hazards also occur in object-oriented systems, with the exception that they result from declarations from within the affected type, whereas in aspect-oriented systems the declarations originate from an aspect outside of the affected type.

M. Marin et al. [43] developed a more detailed fault model in which they divided the faults in three categories: faults in inter-type declarations, in pointcuts and in advice. They distinguish the following faults:

- Inter-type declarations
  - Wrong method name in introduction
  - Wrong class name in introduction
  - Inconsistent parent declaration
  - Inconsistent overridden method introduction
  - Omitted parent interface
- Pointcuts
  - Wrong primitive pointcut
  - Errors in the conditional logic combining the individual pointcut conditions
  - Wrong type, method, field, or constructor pattern in pointcut
- Advice
  - Wrong advice specification
  - Wrong or missing proceed in around advice
  - Wrong or missing advice precedence

  - Advice code causing a method to break its class invariant or to fail to meet its postcondition

Additionally, they describe test adequacy criteria for the implementation under test. For inter-type declarations, they advice achieving normal method coverage goals, plus some extra criteria for polymorphic methods. Pointcut adequacy criteria should be based on traditional boundary testing for wild cards in the pointcut signature, and for multiple conditions in pointcuts [5]. Finally, advice should be exercised at least once for each join point it advices, and for the advice code itself the method coverage goals apply again. For abstract aspects whose pointcuts do not refer to particular joinpoints, a stub application should be made on which the adequacy criteria can then be applied.

J.S. Bækken et al. presented a number of fault categories and a notation style for faults in a fault model [4]. They distinguish the following fault categories:

- Incorrect patterns
  - type pattern faults, e.g., faults in package/class name patterns, wildcards, or specifying || instead of &&
  - method pattern faults
  - constructor pattern faults
  - field pattern faults
  - modifier pattern faults
  - declared-type pattern faults
  - identifier pattern faults
  - argument list pattern faults
  - throws pattern faults
- Incorrect choice of primitive pointcut, e.g., `call` vs. `execution`
- Incorrect matching of dynamic circumstances: involves using `target`, `this`, `args`, `if`, `cflow`, `cflowbelow`
- Incorrect pointcut composition: using the ||, && and ! operators incorrectly, e.g., mixing up || and && between two pointcuts

In his master's thesis report [3], J.S. Bækken proposed a detailed fault model for pointcuts, including some advice fault types. Bækken distinguishes three possible effects that can result from faults in pointcuts:

1. A *positive selection error* (PSE) is present at a join point if, after evaluating some pointcut expression, the expression decides to select the join point, and the join point was not intended to be selected by that pointcut expression.
2. A *negative selection error* (NSE) is present at a join point if, after evaluating some pointcut expression, the expression decides not to select the join point, and the join point was intended to be selected by that pointcut expression.

3. A *context exposure error* (CEE) is present at a join point if, after evaluating some pointcut expression of a named pointcut or a piece of advice, a formal parameter on the left-hand side of the pointcut (or advice) contains an incorrect value.

The entire listing containing all faults that Bækken presents can be found in appendix B on page 91. We give an excerpt of this list below:

3.1.1 Faults in pointcuts

Pointcut fault types:

- Incorrect Pointcut Name: incorrect primitive pointcut
- Incorrect Pointcut Argument(s): includes incorrect patterns
- Incorrect Pointcut Composition: &&, || and ! operators, referencing other pointcuts

Bækken also listed which of the three effects could occur for each specific fault.

3.1.2 Faults in advice

Advice fault types:

- Incorrect Advice Specification
- Incorrect Advice Body

Bækken's fault model does not yet specify faults for inter-type declarations, advice precedence, side-effects of the if-pointcut, aspect instantiation and/or aspect inheritance.

3.2 Adequacy criteria

As mentioned in section 2.2.3 on page 11, an adequacy criterion specifies the elements of an Implementation Under Test (IUT) to be exercised by a test strategy [44]. Some attempts have already been made at defining adequacy criteria for aspect-oriented programs. M. Marin et al. [43] listed a number of adequacy criteria per language element type:

Method introduction: Existing coverage goals apply. For polymorphic method introductions, use the `all-receiver-classes` and `all-target-methods` criteria [39].

Super declaration: All possible polymorphic calls should be covered, thus all `call sites` within the rest of the application have to be covered.

Pointcuts Two kinds of criteria:

Primitive pointcuts: An arbitrary join point will normally do, but when wild cards are involved we can apply boundary testing techniques (one in-point, one out-point).

Compound pointcuts: Every relevant condition combination should be tested.

Advice: Branch/statement coverage goals can be applied, and additionally each the advice should be activated at each captured join point at least once.

Ceccato et al. [11] described two types of coverage, `designator coverage` and `composition coverage`.

Designator Coverage aims at faults in pointcuts that contain conditions on the execution stack. Designator Coverage can be described as follows: Given a pointcut designator D, let pointcut designator E be D after removing all dynamic conditions. Now, the k-th order designator coverage for D is satisfied when all possible execution stacks which include 0 to k instances of all joinpoints captured by E have been exercised. The usefulness of this criterion, however, remains limited as the authors already indicate that the detection of some of these execution stacks can be shown to be undecidable.

Composition coverage aims at fault regarding missing aspect precedence. This coverage criterion requires testing all possible composition orders for aspects which have at least one data dependency.

3.3 Data-flow testing

Data-flow testing is an implementation-based testing technique that identifies test cases by monitoring the flow of data in a program. It "tests how values which are associated with variables can affect the execution of the program" [50], which can find additional errors that specification-based testing may overlook. It results in test cases that test variable assignments by extracting data-flow paths that tracks variable assignments to their usage (def-use pairs).

J. Zhao et al. [50, 49] propose a data-flow based unit testing approach for aspect-oriented programs, in which they apply three scopes of testing for each aspect and class:

- intra-module testing for individual modules
- inter-module testing for a module, along with other modules it calls
- intra-aspect or intra-class testing for modules that can be called from outside the aspect or class

In this case a module denotes a method, which may or may not contain introduced advice or be wholly introduced by an aspect.

J. Zhao et al. later apply this technique to select which test cases should be rerun after a refactoring step of an aspect-oriented program [51].

3.4 State-based testing

R. Binder describes a specification-driven testing pattern for object-oriented state machines based on the FREE state model [5]. Xu et al. have extended this pattern for usage on aspect-oriented state machines [47, 46]. Because the proposed Aspectual

State Model (ASM) is an extension of the object-oriented FREE state model with aspect-specific messages and states, this technique can easily be applied to a development process in which the base classes are implemented and tested before aspects are added. In the ASM, the FREE model for the base classes is extended with extra state transitions and states that have been added by aspects. State transition edges may also fork or be removed as a result of advice that is applied before, after or during the state transition. The approach can be used incrementally (although this is not required), first testing the base model before applying aspects to it. The test cases for the base model can then be extended to account for aspects, but some cases may have to be adapted.

3.5 Control flow based testing

J. Zhou et al. describe an initial approach towards an aspect testing methodology [52]. They propose an incremental approach in which the regular classes are tested first, after which the aspects are individually woven into the code and tested (the aspect classes are considered to be largely independent since they represent orthogonal concerns). After this, there should be an integration test-like step in which multiple aspects are woven into the code and tested, and the final step tests the whole system, including all woven aspects. Test adequacy is measured by control flow analysis, where an aspect reaches complete test coverage when all methods that the aspect advices have been invoked at least once by any testcase. According to M. Marin et al. this criterion is inadequate since coverage of the advice code should also be measured [43].

3.5.1 Model checking

One way of verifying program correctness is by applying model checking, which is a statical code analysis technique that algorithmically verifies a program. Model checking operates on software design rather than its implementation. It can verify whether the design conforms to the specification. N. Ubayashi and T. Tamai [42] and H. Li et al. [25] attempt to apply this technique to aspect-oriented programs. Ubayashi and Tamai propose model checking technique for AspectJ programs, in which they simply weave asserts into the source code that can subsequently be checked using existing model checking techniques. Li lays out a compositional verification methodology, which does not require re-verifying properties of individual features (concerns) on the composed system. Both authors raise concerns about the scalability of this approach however, associated with the state space explosion problem that arises when model checkers encode the data values.

Chapter 4

Testing Aspect-Oriented Programs

This chapter covers most of our theoretical findings regarding the definition of formal testing approaches for aspect-oriented programs. As we have shown in the previous chapter, there has yet been only little progress in this area. For this reason we have made findings from start will present these in a bottom-up fashion: we start out with an aspect-oriented fault model, followed by the definition of a number of adequacy criteria for aspect-oriented programs, and finally we present a number of grey-box testing strategies that targets the faults from our fault model.

4.1 Fault Model

Finding a good testing strategy starts with finding a good fault model. As we have already discussed in section 2.2.2 on page 8, a fault model can help us find effective test cases by providing us with those faults that are most likely to occur. As object-oriented fault models have already been researched thouroughly, we will specifically look at those faults in aspect-oriented programs that do not occur in object-oriented programs.

4.1.1 Using object code

The AspectJ weaver translates an aspect-oriented program into object-oriented bytecode which can be used by the JVM. This means we could skip trying to find aspect-oriented specific tests, and instead focus our efforts on the generated object-oriented model, applying OO-specific tests to the generated class files. This approach, however, is not preferred, at least for the following reasons:

- **It is technically difficult:** When a test fails, we want to know what part of the source code is responsible for the failure. In order to find the source code belonging to a specific portion of bytecode, we need to know to a certain extent how the compiler translates the source code into bytecode. This may be very difficult to determine since the AspectJ language does not specify this mapping, which means that any AspectJ compiler is free to make its own decisions, and subsequent versions of a compiler might also change the way they operate. So finding the source code that belongs to a certain part of bytecode can only be described within a context of a specific version of a specific compiler. And then

it might not even be possible to backtrack each failure to its source, since the compilation process might have discarded the information necessary to do so: compilation is not necessarily reversible.

- **It is worse at finding likely faults:** As we already emphasized in section 2.2.3 on page 11, attempting to use object code in a coverage model has not proven to exhibit any advantages over using source code coverage. This also applies for aspect-oriented code, in which the cognitive distance between source code and object code is even greater.

  Secondly, the object-oriented model contains code that was generated by the compiler. Object-oriented test patterns would also target this code, while this should probably be skipped from testing as it should be considered correct: testing compiler correctness is not the job of an application tester. Testing this generated code thus is a waste of effort, which as a consequence is not applied to finding more likely faults.

4.1.2 Fault listing

The fault model we we are going to derive our tests from is based on the existing fault models that have already been developed for aspect-oriented programs (detailed descriptions of these can be found in section 3.1 on page 19 and appendix B on page 91). We have attempted to integrate the fault hazards found in this prior work, including a few of our own minor additions.

This fault model is not intended to be exhaustive. Apart from the fact that a fault model only contains likely faults and would fail to serve its purpose if it would contain all possible faults, creating test patterns is also an iterative process and fault models continuously evolve as part of this process. This means that a fault model will never be finished. When, for example, a design methodology changes, this may cause new fault hazards to emerge and/or old fault hazards to subside, thus (partly) invalidating the fault model.

The fault model we have applied follows below. We have listed the specific faults per language element.

- Faults affecting aspects

  a1: Wrong Aspect instantiation

- Faults affecting inter-type declarations

  b1: Wrong method signature in introduction

  b2: Wrong class name in introduction

  b3: Omitted parent interface

  b4: Wrong scope

- Faults affecting pointcuts

  c1: Incorrect pointcut strength (PSE, NSE - as explained in section 3.1 on page 21)

c2: Context exposure error

- Faults affecting advice

  d1: Wrong advice specification

  d2: Wrong or missing proceed in around advice

  d3: Trying to modify arguments or return values in before or after advice

  d4: Wrong or missing advice precedence

4.1.3 Detailed fault explanations

Faults in aspects

a1: Wrong Aspect instantiation Aspect instantiation behavior may not always be apparent, especially for the less experienced aspect programmer. Aspects are instantiated automatically rather than by using new expressions. Aspects can be instantiated statically (this is the default instantiation), per object, or per control flow, as specified in the aspect declaration. Subaspects inherit the parent instantiation type per default, although the instantiation type may be overridden by the subaspect. Nested aspect are always statically instantiated.

Wrong aspect instantiation could result in a number of effects:

- Advice may not execute as often as expected: per-object and per-control-flow aspects take a pointcut as an argument, which controls their lifetime. Aspects only match joinpoints while they are alive, and per-object aspect only matches joinpoints on the object it has been declared upon (and thus will also not match any join points from static contexts), while per-control-flow aspects only match joinpoints within that control flow (which limits it to a single thread).

- The `aspectOf()` method might return an unexpected aspect instance, or throw an `org.aspectj.lang.NoAspectBoundException`. For instance, when a per-control-flow aspect is declared on a recursive method, the `aspectOf()` method will return a different aspect instance for each recursive call, while a programmer might expect a single aspect instance to handle the entire recursive callstack.

- Circular dependencies may occur: in the aspect in listings 4.1 on the following page and 4.2 on the next page, for example, the before advice should execute before any constructor, but the advice constructor itself should be executed in order for the before advice to run.

- Changing the aspect instantiation of an abstract aspect might change unintendedly change the instantiation rules of its subaspects.

- Since each aspect instance has its own state (much like classes), wrong instantiation rules could introduce faults on all computations that depend on the aspect state.

```
1 public class Client {
2     public static void main(String[] args) {
3         Client c = new Client();
4     }
5 }
```

Listing 4.1: An aspect containing an instantiation fault - base class (adopted from the AspectJ programming guide [2])

```
1 aspect Watchcall {
2     pointcut myConstructor(): execution(new(..));
3
4     before(): myConstructor() {
5         System.err.println("Entering Constructor");
6     }
7 }
```

Listing 4.2: An aspect containing an instantiation fault - aspect (adopted from the AspectJ programming guide [2])

Faults in inter-type declarations

Inter-type member declarations are very similar to 'normal' member declarations, the main difference being the fact that the declaration can be implemented in an aspect, and thus outside of the class, separating the core code from some concern-related code. We believe the main new fault hazards of inter-type declarations can be found in this enlarged code granularity: a developer could more easily overlook a typing error if related code is placed further apart. Since the code that is placed further apart should represent logically discriminate features, the different code sections should also implementation-wise be loosely coupled (e.g., few method calls from the aspect code to the base code), and we would thus not expect these types of faults to occur as frequently as some others from our fault model, with an exception for the last type (b4 on page 26). We will now proceed with the detailed explanations of the faults we have distinguished in inter-type declarations.

b1: Wrong method signature in introduction The signature of an inter-type method declaration is incorrect, resulting in a missing or unanticipated method override. This can be a fault in its modifiers or identifier, or in its parameter, throw value or return types. Example: `public void MyCommand.undoCommand()` instead of `public void MyCommand.undo()`.

b2: Wrong class name in introduction The class name in the signature of an inter-type method declaration is incorrect. This results in the method body being inserted in the wrong place in the class hierarchy. Example: `public void OtherCommand.undo()` instead of `public void MyCommand.undo()`.

b3: Omitted parent interface An aspect provides a certain class with an implementation of a specific role, but fails to declare the class that specifies the role as its

parent. The declared methods of the affected class now stand on their own. Example: the `public void MyCommand.undo()` is provided by the aspect, but `declare parent:MyCommand implements Undo` is not (and the `Command` class also does not implement `Undo`). We do not expect this fault to occur frequently, since it will usually result in a compile error. Library classes and classes that are accessed through reflection might still be affected, though.

b4: Wrong scope The keywords `private` and `protected` in inter-type declarations relate to the *Aspect* rather than the class they are declared on. Also, the default visibility in aspects is `protected` rather than `package` (the latter of which does is not available in aspects). Wrong assumptions about ITD scope can lead to inaccessible methods in library classes, or worse, execution of the wrong method body if the class specifies a method with the same signature but a different body. We expect this fault to occur especially for developers with little experience in AspectJ, and in cases where code is refactored by moving existing methods to aspects.

Faults in pointcuts

c1: Incorrect pointcut strength As Koppen and Stoerzer noted [24], pointcuts are *fragile*: they select joinpoints by relying on naming conventions, resulting in possible joinpoint matches within future code or client code. When a pointcut is defined either too strong or too weak, advice that relies on this pointcut can fire at join points where it *is not* intended to or fail to fire at join points it *is* intended to fire on. Because of this, pointcuts should select a clearly defined set of join points. For example, let's take the `policyStateUpdate` pointcut from the Eclipse Aspectj book [12]. This pointcut should capture all join points where the policy state for an insurance policy class is updated. An early version of this pointcut,

```
1 pointcut policyStateUpdate(PolicyImpl aPolicy) :
2   execution(* set*(..)) && this(aPolicy);
```

is too strong as it fails to capture the `addClaim` and `removeClaim` methods because of the wrong assumption that the state is only changed in methods that start with `set`. The updated (weakened) pointcut,

```
1 pointcut policyStateUpdate(PolicyImpl aPolicy) :
2   (execution(* set*(..)) ||
3    execution(* addClaim(..)) ||
4    execution(* removeClaim(..)))
5   && this(aPolicy);
```

does the job of capturing the intended join points in `PolicyImpl`. This pointcut could, however, be too strong for future versions of `PolicyImpl` or any of its subclasses, e.g., if a state-changing method like `addPaymentIncident` would be added.

c2: Context exposure error Program context exposed by pointcuts may include wrong object references or null values for non-primitive types, or wrong values resulting from a wrong reference for primitive types. For example, the pointcut

```
1 pointcut mutatorCall(Object o): call(* set*(..)) && this(o);
```

exposes the *caller* in all mehods calls to methods starting with 'set', while the pointcut

```
1 pointcut mutatorCall(Object o): call(* set*(..)) && target(o);
```

exposes the *callee*. In both cases the exposed object can be null, respectively when the method call originates from a static context and when the called method itself is static.

Faults in advice

d1: Wrong advice specification The signature of a piece of advice is incorrect. Specific faults in the advice specification include:

- wrong advice type (before, after (returning/throwing), around)
- wrong type(s) in advice (advice formals, return type in around advice and type in after returning/after throwing formal)
- wrong pointcut specified or referenced

These can lead to the following errors:

- When a wrong advice type is used, this can lead to unexpected behavior since the advice code may be executed at an unexpected point of execution, e.g., when after advice is used instead of after returning advice, the advice will also (unintendedly) run when an exception is thrown.
- Using wrong type(s) in advice may result in a ClassCastException (especially when using `Object`).
- Since the type(s) used in the advice signature can filter out certain joinpoints captured by the pointcut (effectively adding additional constraints to the pointcut), incorrect pointcut strength may also occur here. One notable example is using an `Object` return type in an around advice, which may (unintendedly) also capture `void` methods.
- Advice is also prone to context exposure errors - it can expose the wrong objects from a pointcut, e.g., when using `object (after(Object o):pointcut(o,*)` instead of `after(Object o):pointcut(*,o))`

d2: Wrong or missing proceed in around advice When the `proceed` statement in an around advice contains wrong arguments or is missing (entirely or within a particular control flow). A wrong `proceed` statement can include a wrong type (leading to to a ClassCastException), when the advice specification includes a supertype but a proceed argument contains a certain subtype, and the advice operates on a joinpoint which uses another subtype.

d3: Trying to modify arguments or return values in before or after advice Before and after advice cannot directly affect either a method's arguments or return value. Both before and all three after advice types (after, after returning and after throwing) can expose a method's arguments, but changing their values only affects the code

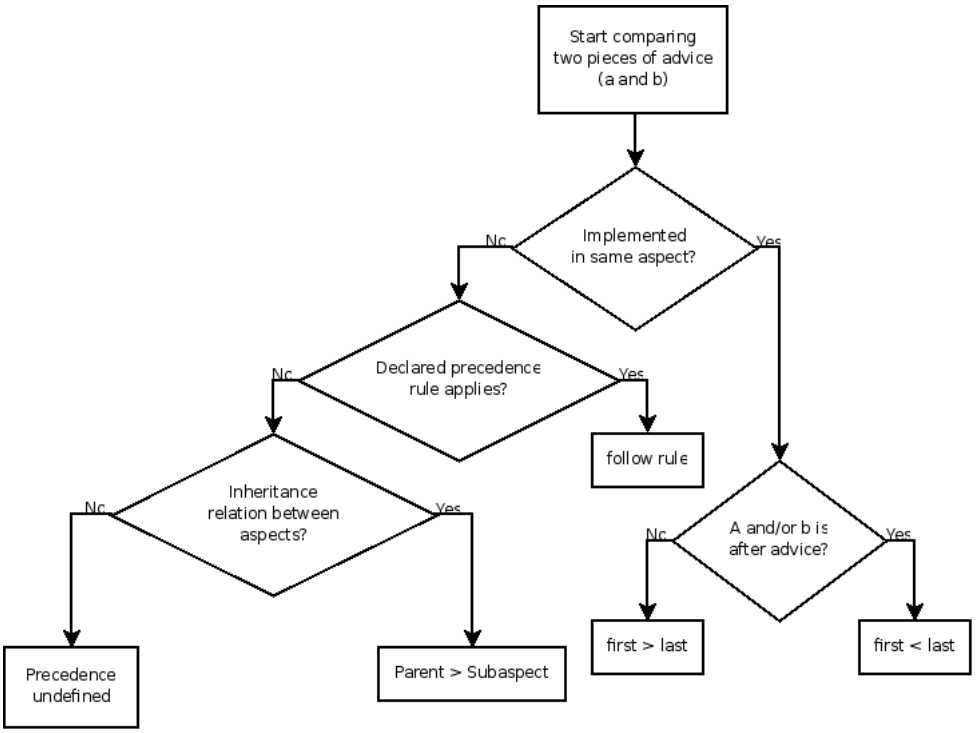

Figure 4.1: Flowchart summarizing AspectJ precedence rules

within the advice: the original values will be passed on to code within other blocks of advice and/or the base implementation. This also holds for

Trying to do so has no effect, though it may seem possible because they can be exposed (only after returning and after throwing advice can expose the return value, which of course in the case of an after throwing advice of course is an exception).

d4: Wrong or missing advice precedence When two or more pieces of advice apply to one join point, the order of execution is determined by their precedence. AspectJ applies a somewhat intricate mechanism for determining advice precedence.

Precedence in AspectJ AspectJ determines the precedence for all pairwise combinations of advice affecting a join point (including combinations of `before` and `after` advice, for which it is not strictly necessary). For these pairs of advice, their precedence can be affected by the advice *types*, the enclosing *aspect* in which the advice blocks are implemented, and the *order* of advice implementations in one aspect. Figure 4.1 summarizes AspectJ's precedence rules. These rules do allow precedence circularities in some cases. AspectJ throws a compile error if this occurs.

At each join point, the order of execution is as follows:

- All `before` advice and `around` advice (up until the `proceed()` statement) is executed, from highest to lowest precedence.
- The base code from the join point executes

- All `after` and `around` advice (from the return of the `proceed()` statement onwards) is executed, from lowest to highest precedence.

Precedence faults Wrong advice precedence can be caused by the following errors:

- A missing advice precedence rule (only for advice defined in different aspects)
- A wrong advice precedence rule (only for advice defined in different aspects)
- Wrong ordering of advice inside an aspect
- A wrong advice type (substituting `after` advice with `around` advice, for example, affects the precedence in relation to all `before` and all other `around` advice defined in the same aspect)

Wrong advice precedence can only cause advice being executed in an illegal order, which in turn can lead to unexpected behavior.

4.2 Adequacy criteria

Section 3.2 on page 22 show us the preliminary suggestions for adequacy criteria in aspect-oriented software. Here we will present a synoptic overview of those criteria we deemed useful, including some of our own contributions. We will, however, start out by explaining a little more about the nature of pointcuts and the resulting limitations for defining aspect-oriented adequacy criteria.

4.2.1 Pointcut goals

In order to be able to understand the applicability of adequacy criteria for aspect-oriented software, we should explain a little more about pointcuts in AspectJ first. We have distinguished three distinct goals for pointcuts in AspectJ :

Join point selection: A pointcut's prime goal, as we have explained before, typically is to select a number of join points from the entire set of joinpoints in a given program.

Context exposure: Pointcuts can additionally expose some parts of the execution context of the join points they select. The `this`, `target` and `args` primitive pointcuts specifically fulfill this goal.

Branching: The final pointcut goal we identified is branching. With branching we mean that a pointcut can decide whether to include or exclude an already-selected join point at runtime based on dynamic information. This essentially is similar to adding an `if(some condition) return;` statement at the start of a piece of advice, but for the fact that AspectJ allows some extra conditions based on revealing extra information of the runtime stack.

Static pointcuts can only be used to select joinpoints. Dynamic pointcuts, on the other hand, can be used for different combinations of these goals:

- The `if` pointcut can be used for both joinpoint selection and branching. Branching seems the most obvious use for `if` pointcuts, for example by letting advice only run when an object is in a specific state, using a construction like `if(theGame.inPlayingState())`.

  However, in some cases the `if` pointcut can also be used for selecting joinpoints based on static conditions. As an example we present one of our own pointcuts, with which we excluded all joinpoints in anonymous classes. Since AspectJ does not include a construct to accomplish this in its type patterns, we had no other choice than to use an `if` pointcut:

```
1 pointcut commandExecuteCheckView(AbstractCommand acommand) :
2     execution(void AbstractCommand.execute())
3     && this(acommand)
4     && !if(acommand.getClass().isAnonymousClass());
```

- The `cflow` and `cflowbelow` pointcuts can be used only for branching, since they only make use of `dynamic` information (the callstack) to determine whether to run advice on a joinpoint.

- The `args`, `this` and `target` pointcuts combined with pointcut and/or advice arguments can, in addition to exposing join point context, be used for join point selection and/or branching goals. Join point selection and branching goals can be applied by using a `Type` pattern instead of an `Id` pattern [2] or by specifying a specific type in the argument part of the joinpoint or pointcut. The `this` and `target` also exclude any join points from static contexts, possibly applying additional join point selection. For example, let's consider the following class:

```
1 public class Foo {
2     public void foo(Object o) {}
3     public void foo(Object o, String s) {}
4 }
```

  and the following three pointcuts:

```
1     pointcut p1(Object o):
2         execution(* Foo.foo(..)) && args(o);
3     pointcut p2(String s):
4         execution(* Foo.foo(Object)) && args(s);
5     pointcut p3(Object o):
6         execution(* Foo.foo(Object)) && args(o);
```

  All three pointcuts expose context information: the method argument. However, the pointcut argument in pointcut p1 also excludes the two-argument `foo` method execution join point from the pointcut, thus also performing a join point selection goal. The pointcut argument in pointcut p2, on the other hand, performs a secondary branching function: each time the one-argument `foo` method executes, the pointcut checks whether the actual type of the method argument is a `String`, and does not capture the joinpoint otherwise. Only the pointcut argument in p3 solely performs a context exposure goal.

4.2.2 Implementation challenges and restrictions

To be able to implement some aspect-oriented adequacy criteria, we need to be able to distinguish which of the three goals a certain pointcut expression serves. If, for example, we were to measure the branch coverage for a given method, including the branching statements of the advice applied to that method and the branching expressions in the pointcuts that the advice applies to, we would have to exclude the join point selection expressions in the pointcuts[1] while including the branching expressions.

We determined three general ways of distinguishing the pointcut goals:

- **Modifying the AspectJ language**: A few minor changes to the AspectJ language could allow only a single goal for each primitive pointcut:
  - Pointcut selection or branching goals could be disallowed entirely for the `this`, `target` and `args` primitive pointcuts: the same effects can also be accomplished by adding additional constraints or primitive pointcuts to the pointcut.
  - By adding a second `if` pointcut (named otherwise, of course) that only has access to static joinpoint information, the original `if` pointcut can be used solely for branching purposes while the new `if` pointcut can be used solely for pointcut selection.
- **Goal marking**: In order for some specific adequacy criteria to be calculated, we could require some pointcuts to be marked as to reveal their actual goals somehow.
- **Automatic goal detection**: To some extent, it might be possible to automatically detect the pointcut goals. As for the `args`, `this` and `target` primitive pointcuts, this is quite trivial, and has in fact already been accomplished by the AJDT[2] project. The `if` pointcut, however, poses a greater challenge since it may contain an arbitrary expression. Detecting which parts of this expression only access static information and which parts need dynamic information may be more difficult.

Even if a method would be available for distinguishing the goals of a particular pointcut expression, some related details still remain to be addressed. These follow below. The second and third additionally show that automatic goal detection is not a viable option for distinguishing pointcut goals.

Pointcut and advice argument types

As we mentioned before, both pointcut and advice arguments combined with `args`, `this` and/or `target` pointcuts can, in addition to exposing join point context, be used for join point selection and/or branching goals. Both this and the fact that a block of advice might extend an existing pointcut or define one of its own means that adequacy

[1] We can also fill them in, since these expressions will always carry the same truth value at the tested join point

[2] `http://www.eclipse.org/ajdt/`

criteria that measure coverage per pointcut should in these cases have to measure coverage separately for each block of advice (at least for the blocks that extend or define pointcuts).

Goal separation

Pointcuts can contain a mixture of goals. Adequacy criteria specifically measuring one pointcut goal need to disregard the parts of the pointcut expression that serve other goals (effectively measuring coverage over a simpler expression). For compound pointcuts consisting solely of primitive pointcuts that serve a single goal each, this process is trivial: simply remove (disregard) all primitive pointcuts that serve a goal other than the one tested for from the pointcut expression[3]. We also believe it would be be achievable for the `args`, `this` and `target` primitive pointcuts and pointcut or advice arguments. However, we do not believe this to be true for applying introspection of the expressions contained by an `if` pointcut: in order to be able to disregard the parts of the expression that serves other goals, we would have to gain insight into arbitrary Java expressions, a task that we estimate would prove very difficult to impossible. For this reason we do not believe automatic goal detection to be a viable solution for distinguishing the pointcut goals.

Detecting selected pointcuts

As M. Marin states [43], "it is natural to insist that each join point at which the advice is activated is exercised". However, we do not believe it will be possible to implement an adequacy criterion that measures this: we would have to be able to determine exactly which joinpoints are captured by a pointcut. As we just stated that we do not believe that we could automatically separate the goals for an arbitrary `if` pointcut, which sadly also implies we cannot automatically determine the joinpoints captured by an arbitrary `if` pointcut.

4.2.3 Aspect-oriented adequacy criteria

The restrictions and challenges on the implementation of adequacy criteria aside, we have identified a number of possible adequacy criteria to be used for aspect-oriented programs.

- Adequacy criteria measuring test coverage of join point selection goals
  - **Join point execution coverage**: All join points of the IUT have been exercised at least once. This criterion does not measure whether or not any advice is executed at the join points.
  - **Condition coverage for pointcut selection**: M. Marin gives some pointers towards defining an adequacy criterion for measuring the pointcut selection boundaries in pointcut expressions [43]. We would like to note, however,

[3] That is, assuming that pointcut expressions serving different goals are only combined through conjunctive statements (`&&`), which should be the case if we consider all branching expressions to be able to assume both truth values at each joinpoint

that this method requires disregarding all pointcut expressions that do not serve a pointcut selection goal.

- **Joinpoint-advice execution**: This conceptually trivial coverage criterion requires that a block of advice is at least executed once at each captured joinpoint. For this coverage criterion we would note that in order to implement it, we should be able to detect the selected pointcuts.

- Adequacy criteria measuring test coverage of branching goals

  - **Line, branching and path coverage**: These coverage criteria can be applied per block of advice or per join point. If we measure per advice, we only measure the branching statements from the advice pointcut (including referenced pointcuts), and all statements in the advice block itself.

    When we measure the coverage per join point, we have to include all lines/branches/paths of the join point itself, all advice defined upon it and the branching statements from the advice pointcuts. For measuring path coverage, we would advise to consider the order of advice for which the precedence is undetermined also to be undetermined for (and excluded from) the to-be-covered execution paths. This means that while the execution order of these blocks of advice will eventually determined by the compiler, we would advise that only a subset of these paths should be required for full path coverage.

    It might not be very practical to strive for full join point full path coverage in all cases (especially when there are no data dependencies between the join point and advice code), but it could prove insightful in some cases (e.g. when the code sometimes fails at a particular join point, one of the branches that have not been covered by the test suite on that particular join point would probably cause the error)

We do not believe any additional adequacy criteria would be necessary for context exposure goals. Furthermore, we do not claim or assume this list to be either sufficient or exhaustive.

4.3 Grey-box testing

In section 2.2.2 we explained the difference between white-box and black-box tests. While the adequacy criteria we have mentioned in the previous section could, when implemented, be used as white-box tests (measuring the fitness of the test suite), in this section we will describe some actual tests to be used in a test suite for an aspect-oriented program.

The testing techniques we will discuss in this section are what we call *grey-box* tests. With this we mean that we have applied a mixture of black-box and white-box techniques. A typical white-box property is, for example, the fault model from which we derived these tests, described in section 4.1. This is a specific fault model, based on the structure of an AspectJ program rather than its specification. We are, however, testing against the intended instead of the actual behavior, and in some cases we rely on existing black-box testing techniques.

4.3.1 Tests for faults in aspects

Fault a1: Wrong Aspect instantiation
Testing: The number of possible effects resulting from wrong aspect instantiation are numerous as well as diverse, pretty much disallowing an effective general risk-based testing strategy targeting these effects. It is, however, possible to observe actual aspect instantiation by adding an aspect to the test suite that monitors the tested aspect's constructor, and/or by using the `aspectOf()` method. We would advise for per-object advice to check when it is instantiated, testing both the *absence* of the aspect just before it should be instantiated as well as the *presence* just after. For per-control-flow aspects we would advise to perform these two checks both on instantiation and destruction, constituting four test cases in total.

4.3.2 Tests for faults in inter-type declarations

The few new bug hazards resulting from inter-type declarations can be covered adequately by providing the declaree class with a responsibility-driven test set.

Fault b1: Wrong method signature in introduction
Testing: As the method that was intended to be overridden is in fact not, a responsibility-driven test case for the overridden method should reveal this fault.

Fault b2: Wrong class name in introduction
Testing: Same as testing fault b1 on page 26, a responsibility-driven test case for this method should reveal it.

Fault b3: Omitted parent interface
Testing: Simple, cast the class as an instance of the interface. However, when the parent interface is omitted, the test is also likely to be omitted. This test might still useful to guard against removal of the parent interface in possible future changes.

Fault b4: Wrong scope
Testing: When a wrong private or protected method is called from within the code, this should alter the caller method's behavior, which should be detected by the responsibility-driven test case for the caller. Faults in library methods (with protected visibility) should also be covered by the responsibility-driven test suite for the declaring class.

4.3.3 Tests for faults in pointcuts

Fault c1: Incorrect pointcut strength
Testing: Pointcuts select a number of join points from the domain of all join points. We can apply a domain testing approach, testing the boundaries of the domain by picking a number of join points close to the domain boundary and checking whether the pointcut includes or excludes these join points as expected.

But how can we find join points close to the boundary? In an ordinal domain (e.g., scalar types), it's easy to define the distance of two test values (the number 5

```
1 pointcut commandExecuteCheckView(AbstractCommand acommand) :
2     execution(void AbstractCommand.execute())
3     && this(acommand)
4     && !if(acommand.getClass().isAnonymousClass());
```

Listing 4.3: Sample pointcut

```
1 JoinPointTypeIsExecution /* 1. JoinPoint.kind */ &&
2 methodReturnsVoid /* 2. Joinpoint.signature.returnType */ &&
3 noMethodArgs /* 3. Joinpoint.signature.method */ &&
4 methodIsNamedExecute /* 4. Joinpoint.signature.method(2) */ &&
5 declaringTypeIsAbstractCommandOrSubclass
6     /* 5. JoinPoint.declaringType */ &&
7 declaringTypeIsNotAnonymous  /* 6. JoinPoint.declaringType(2) */
```

Listing 4.4: Abstract class invariant for pointcut 4.3

is further away from the number 3 than the number 4 or 2), but for more complex domains like the set of all joinpoints, this is not as easy. We can however follow Binder's recommendations for domain testing objects [5, p. 407-410], by constructing an abstract state for our pointcut. We can look at the joinpoint class structure for this, and consider the pointcut a function that accepts a JoinPoint as an argument and returns a match or non-match. An abstract state is constructed by making a function that 'evaluates a state invariant expression and returns true or false' [5], only focusing on values that matter most. We will elaborate this strategy using an example.

Pointcut strength testing elaboration and example

Listing 4.3 shows a sample pointcut in AJHD, for which we will now determine an abstract class invariant. For the first statement, `execution(void AbstractCommand.execute())`, it is not very obvious which boundaries to test for. Looking at the JoinPoint structure, however, we can find a number of attributes that play a part here. The resulting abstract class invariant can be found in listing 4.4.

As the conditions that consitute the abstract class reduce the domain boundary to a combinatorial expression combining properties expressed as boolean values, it would seem most practical to use the Each-Condition/All-Conditions test selection criterion[4] [5, p. 155-157] rather than the 1×1 domain testing strategy[5] [5, p. 410-412], which is geared towards testing scalar values rather than boolean values. We would, however, recommend a mixed strategy: some of the abstract properties can be treated as scalar values rather than just boolean values. For example, the declaring type property of being a subclass of abstract command including abstract command itself, we could imagine each class being on the scale abstractCommand-class-hierarchy, considering direct super- and subclasses of `AbstractCommand` being close to the domain

[4] Simply put, this criterion prescribes $n+1$ testcases for n boolean variables, - one testcase for each variables in which it attains a `false` value while keeping the other ones `true`, and a final all-true-values testcase (we assume only && logic here, but it can be applied to || logic or combinations as well).

[5] Simply put, this criterion describes 2 testcases for each domain boundary: one testcase on the boundary and one just off, i.e., for an integer $x >= 0$ we would test $x = 0$ (on) and $x = -1$ (off).

border, and indirect super- and subclasses (x-degree super- and subclasses) being further away. This way we can come up with typical values, on- and off-points. The declaring type being anonymous or not, however, should clearly be treated as a pure boolean value (there is no notion of an anonymous class being more anonymous than another anonymous class).

To accomplish this, we start out by dividing the different properties into two categories - boolean values, for which we will apply the Each-Condition/All-Conditions selection criterion, and scalar values, for which we will devise tests following the 1×1 domain testing strategy. For some properties it may not entirely clear in which category they should fall. For example, there are a number of kinds of joinpoint. However, since we cannot determine an ordering between these kinds of joinpoints, we consider this a boolean.

The division of our abstract class invariant properties:

1. `JoinPoint.kind == execution` - boolean
2. `Joinpoint.signature.returnType == void` - boolean
3. `Joinpoint.signature.method.args == none` - boolean
4. `Joinpoint.signature.method.methodname == ''execute''` - boolean
5. `JoinPoint.declaringType <= AbstractCommand` - scalar (x-degree super- and subclasses)
6. `JoinPoint.declaringType != Anonymous` - boolean

For each boolean value, we create one testcase in which that boolean value evaluates to `false`, and the other boolean values evaluate to `true`, while we pick a typical value (in-point) for each scalar type. For each scalar type, we pick one on-point and one off-point in two separate testcases, for both of which the boolean values should evaluate to `true`, while we pick a typical value (in-point) for all other scalar types. Finally, if we are only testing boolean values, we should add one testcase in which all boolean values evaluate to `true`. This is not necessary if we are also testing scalar types as the in-point testcases already cover this. The total number of testcases now generally comes to (boolean properties) + (scalar properties*2) (+1 if only testing boolean properties).

For our example, this comes down to $5 + (1 * 2) = 7$ testcases.

1. `JoinPointTypeIsExecution == false`: Here we can use a call joinpoint on the `execute` method. Using an aspect, we can check whether the pointcut has already fired at the call joinpoint (which should not be the case).
2. `methodReturnsVoid == false`: Here we should run an execute method that does not return a null value. As adding such a method for testing purposes would result in a compiler error, we consider this testcase infeasible.
3. `noMethodArgs == false`: Run an execute method that takes 1 argument.
4. `methodIsNamedExecute == false`: Run a method that is not called execute on a command subclass.

5. `declaringTypeIsAbstractCommandOrSubclass on-point`: Here we should run the `AbstractCommand.execute()` method, but as `AbstractCommand` is abstract, the only way to do this is to add a subclass that does not override the default `execute()` implementation.

6. `declaringTypeIsAbstractCommandOrSubclass off-point`: The nearest off-point would be a direct superclass of `AbstractCommand`, but `AbstractCommand` is a top-level class which only implements interfaces. The closest match would probably be any class that is not a subclass of `AbstractCommand` but does implements its interfaces.

7. `declaringTypeIsNotAnonymous == false`: Run the execute method on an anonymous subclass of AbstractCommand.

The resulting tests are listed in listing C.17 on page 124. To be able to observe whether the pointcut either hit or missed a joinpoint, we have added a custom aspect to the code that operates on the joinpoint under test. This aspect can be found in listing C.16 on page 123. We should note, however, that it is not always possible to test a pointcut by applying additional advice to it: restrictive pointcut visibility or pointcuts implemented in advice specifications can prevent aspects in the test set to access them. In these cases, the tests can only verify pointcut hits by observing the behavior of advice in the IUT that has been applied to these pointcuts.

(Un)nessecary constraints

As we found testcase #2 to be infeasible, the corresponding constraint could have been left out without affecting the code. We have had to add custom command code for implementing tests #3 and #6, which would possibly indicate further unnecessary constraints.

In the case of #3, this argument seems to hold: the pointcut could also would have worked fine if it was absent - the code does not contain any `execute` methods which take a non-zero number of arguments. For testcase #6, though we needed the additional code to approach the constraint border as close as possible, the constraint would still seem unnessecary: the AJHD code does not contain any methods named `execute` aside from the `AbstractCommand` subclasses.

We should not forget, however, that JHD is a framework: its code has been designed to be extended in third-party applications. This places additional constraints on the pointcuts as it should also select the correct joinpoints in this extended code, which may contain, for example, `execute` methods outside of the `AbstractCommand` hierarchy. This validates the use of these additional constraints after all.

Fault c2: Context exposure error

Testing: We could apply the *design by contract [34]* principle for pointcuts, treating the exposed context as a postcondition for the pointcut, which should as such also be documented by the pointcut. Checking this postcondition, however, is a bit more difficult - we believe the most viable solution would be to check the pointcut postcondition in the advice defined on the pointcut. This does have the unfortunate effect of having to perform the same check in all advice that operates on a single pointcut, however.

4.3.4 Faults in advice

Fault d1: Wrong advice specification
Testing:

- Test the exposed arguments - are they what you expect them to be?
- Attempt to run the advice at least once at each join point selected by the advice pointcut.
- If the advice implements or extends a pointcut, apply the boundary testing technique for incorrect pointcut strength.

Fault d2: Wrong or missing proceed in around advice
Testing: A wrong or missing `proceed` statement should result in unexpected behavior, which should be covered by the responsibility-driven test set.

Fault d3: Trying to modify arguments or return values in before or after advice
Testing: This could be captured by a compiler warning (or by a simple `declare warning` statement), but otherwise it should also result in unexpected behavior, covered by the responsibility-driven test set.

Fault d4: Wrong or missing advice precedence
Testing: We cannot explicitly test whether a number of pieces of advice that should execute in a particular order, also have been implemented to run in this order: when precedence rules are missing, the precedence determined by the compiler might by chance be the correct one. Also pieces of advice are not named, so even when using an `adviceexecution()` join point it is not always possible to determine which piece of advice is actually being executed.

Chapter 5

Application to the AJHotdraw case study

We have chosen the AJHotDraw (AJHD) project as a case study for our aspect-oriented testing techniques. The AJHD project is an aspect-oriented refactoring of the JHotDraw (JHD) project, which is commonly used as an aspect mining case study [10, 38, 40, 48]. In this chapter, we will start out by outlining the structure of the JHD and AJHD projects. We will explain a bit about the refactoring process in AJHD, and introduce the test framework in the TestJHotDraw(TJHD) subproject of AJHD. Then we will use two of the refactorings in AJHD to show the appliance of our aspect-oriented testing techniques. Finally we will summarize the bugs in that we have found, and which sorts of crosscuttingness we have encountered.

5.1 The JHotdraw codebase

The JHD project has been conceived as a programming exercise and a showcase for demonstrating programming using patterns [18], its initial team containing some profound members from within the software development community, most notably Erich Gamma (co-author of the notable Design Patterns book [18]). For these reasons, we had expected the JHD project to be a prime example of well-designed software, and thus an interesting case for finding improvement possibilities when using aspects. Later on we will see that JHD sadly did not fulfill these expectations. The current version of JHD is 7.08, but as AJHD is a refactoring of JHD version 6.0b1, we will be using this older version.

5.1.1 Key framework classes

As mentioned section 2.4, JHD implements a two-dimensional graphics framework. It contains the framework itself accompanied by a few drawing editors that act showcases for the framework functionality. We will introduce the framework structure by means of a few key interfaces from org.jhotdraw.framework package:

- A drawing is, unsurprisingly, represented by the *`Drawing`* class
- Graphical elements in a `Drawing` are represented by the *`Figure`* class. `Figure` is a Composite [18], and has a `paint` method that paints it.

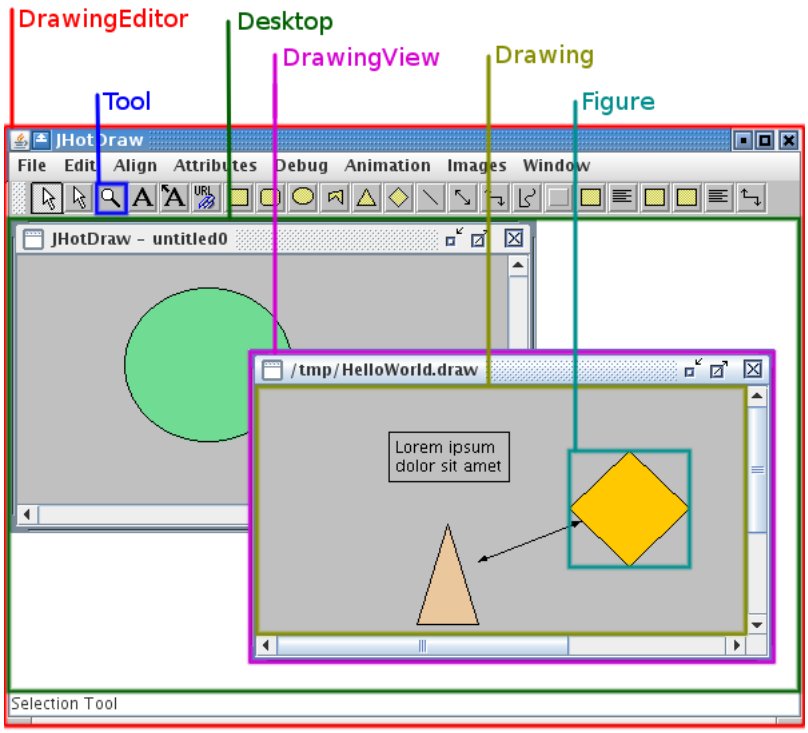

Figure 5.1: The visual representation of a number of key JHD framework classes in `JavaDrawApp`

- A `Drawing` can be rendered by a *`DrawingView`*.
- A *`Tool`* defines a mode of the `DrawingView`. A `DrawingView` forwards all input events to its current `Tool`.
- The *`DrawingEditor`*, which typically defines the drawing window on the desktop, can contain one or multiple `DrawingView(s)`.
- The `DrawingEditor`'s `DrawingViews` are typically, but not necessarily, maintained by an instance of the *`Desktop`* interface.

See Figure 5.1 to see where instances of these interfaces fit in a sample application. We have also included a class diagram illustrating their relations in Figure 5.2.

These classes already exhibit a high pattern density: `DrawingView`, for instance, is an `Observer` which listens to its drawing for changes to be able to redraw when necessary. It also plays the role of the `StateContext` in the State pattern (`Tool` is the `State`), and the `StrategyContext` role of the Strategy pattern for both the update strategy and point constrainer. In this last example, the update strategy defines when to (re)paint different layers of the drawing, and the point constrainer allows the implementation of different kinds of grids in the drawing.

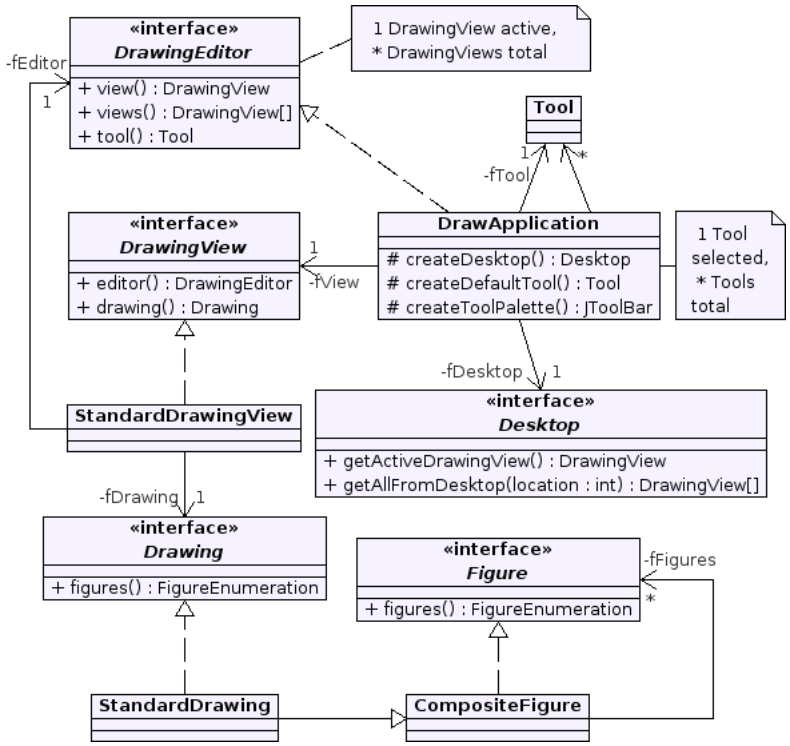

Figure 5.2: Class diagram containing the classes highlighted in section 5.1

5.1.2 Applications in JHD

JHotdraw ships with the following application examples:

NothingApp: This is a basic application containing simple shapes and connectors. It can save drawings in the JHotDraw native and Java serialization formats.

JavaDrawApp: A more complete editor. Compared to the NothingApp it has a number of extra tools, including drag and drop, text area, border and scribble.

SVGDrawApp: This editor is equal to JavaDrawApp, save for the fact that it can only output its drawings in the Scalable Vector Graphics (SVG) format.

NetApp: An application which can be used to build simple networks of nodes and connectors.

PERTApp: A Simple Program Evaluation and Review Technique (PERT) diagram editor.

MiniMapApp: This application seems to be unfinished and/or unmaintained. Running it only results in 'null' being printed on stderr.

5.1.3 JHD fails to live up to our expectations

The rather disappointing performance of the MiniMapApp emphasizes one of the less appealing 'aspects" in JHD: the code has been around for some time, and some parts have received better maintenance than more unfortunate others. The MiniMapApp fails to start as a result of a faulty image URL (the package structure has been renamed in JHD6, but the image URL still contained the old package structure). The image subsequently fails to load, and the resulting `NullpointerException` is caught. The catch clause only prints the message to stderr and omits the stacktrace, thus making debugging a lot harder.

Another limitation in JHD is the lack of documentation. Javadoc comments are generally incomplete, outdated or missing, and all other forms of documentation (e.g., class diagrams, text files containing descriptions of JHD's structure) are scarce and qualitatively even worse. This proves a challenge for proper testing: if we can hardly figure out what the code is supposed to do, how can we verify that it works as it is supposed to? Fortunately we are primarily looking for changes in the behavior, not in verifying the behavior itself. We have, however, already seen that having a complete set of tests for the system before refactoring is a valuable, if not necessary, starting point for doing that. JHD does provide an extensive test suite, but closer inspection reveals that all of the test are still empty and do not actually test anything. This means that these can also not be used as a documentation reference or be reused for testing refactorings. JHD also does not enforce class contracts using assertions. In fact, the JHD source does not contain any assertions at all. In some cases, however, exceptions are used for this purpose instead.

5.1.4 AJHD - refactoring JHD

The AJHotdraw(AJHD) project, also mentioned in section 2.4, is an aspect-oriented refactoring of the JHotDraw project (version 6.0b1), which contains two modules: one for the refactored source (AJHotDraw), and another one (TestJHotDraw) that contains the test cases which can be run on both the original and refactored programs. AJHD specifically contains a number of refactorings based on crosscutting concern *sorts* [30, 29] - generic descriptions of crosscutting functionality that can be classified based on three main characteristics [31]:

- intent of the concern (behavioral, design or policy requirements);
- legacy (non-aspect) implementation idiom;
- (desired) aspect language mechanism that supports the modularization of the sort's concrete instances.

M. Marin et al. have identified and refactored a number of these CCC sorts in JHD, including a number of initial tests in the TestJHotDraw (TJHD) subproject. We have made some minor additions and improvements to the existing refactorings, and most importantly we have applied our testing techniques to expand the set of tests in order to determine whether the applied refactorings did not invoke any behavioral changes.

Next we will explain the test setup in more detail.

5.2 Test framework and general set-up

5.2.1 Intent

The TJHD subproject aims at ensuring behavior conservation between JHD and AJHD. Testing behavior conservation while refactoring is usually achieved by having a test set beforehand and running it both before and after the refactoring. Successful completion of both test runs then indicates similar behavior between the two implementations. In TJHD we have strived to implement a set of tests that can be run on both the JHD and AJHD code. These tests can, beside demonstrating behavioral equivalence, be used for regression testing when applying new refactorings. Being able to run the tests on both systems with a single button or command would arguably be the highest level of automation achievable, so this is what we have tried to accomplish.

Dealing with structural differences in the implementation

Some refactorings require code changes in both the test set and the base code. Even applying a simple OO refactoring like *move method* [17] would at least require changing all method calls to the affected method in both the base code and the tests, but more changes (such as moving some unit tests that exercise the affected method to other test classes) are probably required.

The refactoring itself may also prompt new tests; the refactored implementation may provide new testable elements. For example, the *add parameter*, *extract subclass* and *replace type code with state* can respectively introduce a new parameter, a new method and a new state. Refactorings may remove testable elements in a similar manner, also removing the need for some testcases.

The same reasoning applies for AJHD even though the project applies a different set of refactorings. This means a single test suite does not suffice. Though some tests may be run on both the JHD and AJHD implementation, other tests could either be implemented differently for JHD and AJHD, or only run on one of the two implementations. We have solved this problem by creating different test suites as necessary: one test suite that purely operates on JHD, one that can be run on both implementations, and one solely for AJHD. Since JHD did not have any tests of itself[1], we have had to add tests for JHD in addition to tests for AJHD. Fortunately, the TJHD project already contained a small number of tests when we started on the project, which spared us some work.

Dealing with failing tests

When we started adding tests for JHD, we discovered the project contained a large amount of bugs: our final test set attains a failure rate of around 20% (56 test failures in total) on both JHD and AJHD. AJHD is a *refactoring* project, i.e., it is supposed to change JHD's structure without changing any behavior. This also means that any defects in JHD are carried over to AJHD. We assume the lack of tests in JHD is a major contributing factor of JHD's disappointing code quality.

[1]The JHD actually does ship with a sizeable test suite, but on closer inspection these reveal themselves to be automatically generated unit test which do not actually perform any testing

We could have lowered the test failure rate or completely reduced it to zero by changing the tests to check for the `actual` behavior, but we believe that in that case the tests would not check for the *intended* behavior as they should. As JHD does not specify most of its intended behavior, one could argue that the actual behavior should be considered the intended one (the source code is the best and only documentation JHD provides). There often are, however, a number of reasons to assume otherwise:

- When the intended behavior is obvious: for example, if we delete a figure from a drawing and subsequently undo this delete, we would expect the figure to be restored to its state before the undo took place; when our deleted figure would appear on a different position, for example, we would clearly consider this a bug. Obvious intended behavior can arise in different contexts including differences and similarities with other computer programs, intentions of applied programming patterns, and element names (classes, methods, etc.).

- When a feature is implemented inconsistently in different locations. For example, some, but definately not all `Storable` subclasses include a `private int xxxSerializedDataVersion` attribute which, judging by its name, is intended to be used for comparing the actual class version with a restored object of the same class.

To be able to check for behavior preservation without leaving out the failing tests, it is possible to check whether the same tests fail on JHD and AJHD, or even better, to check whether the tests fail emphin the same way. If they both display the same faulty behavior then we have a better indication for behavior preservation than we would have had if we had left out the tests. To implement this we have added an extra test suite that does not actually perform tests on the sources, but performs a pairwise comparison of each individual test in the test results from the tests that run on both JHD and AJHD. This test suite only reports test success if the compared test results were equal, meaning that either both tests ran successfully, or both failed on the same line of code of the testcase. We believe this is the best comparison we can make, pushing it any further (e.g., by comparing the complete stack trace), we could end up judging structural differences as behavioral ones and thus find false positives.

5.2.2 Implementation

TJHD contains three major JUnit[2] test suites:

- `org.jhotdraw.TestAllAJHD` in `src/ajtest` contains unit tests that can be run only on the AJHD source code

- `org.jhotdraw.TestAll` in `src/test` contains unit tests that can be run on both the JHD and AJHD source code

- `org.jhotdraw.TestCompliance` in `src/test` does not contain any unit tests, but can compare the accumulated test results from the former test suite when it has been run on both JHD and AJHD.

[2] `www.junit.org`

As we were fortunate enough not to encounter any refactorings which required separate test code for both JHD and AJHD, we did not need a test suite for JHD-specific tests. We did need the `TestAllAJHD` test suit for some aspect-specific tests.

We used apache ant[3] to be able to easily run all of the available tests. The first and second test suite can also be run without the use of ant, but successfully running the `TestCompliance` tests is a more complicated process without using ant. This is mainly due to the fact that the `org.jhotdraw.TestAll` test set must be compiled, woven and run separately for JHD and AJHD to avoid name collisions.

Ant targets

The ant buildfile lists a number of targets. The documented targets provided by the `ant -projecthelp` command are:

```
Main targets:

 build          Build the tests, weaving them with the JHD and
                AJHD code
 clean          Clean the generated output
 cleantests     Clean Test reports
 doc            Create JavaDoc documentation
 test           Run the JUnit tests. Set the property test.suite to
                run only a specific suite, for example 'ant test
                -Dtest.suite=org.jhotdraw.aspects.invariants.Invar-
                iantCheckTest' only runs the invariant check test
 test.coverage  Run the tests and measure code coverage.
 zip            Source distribution of the test sub-project
Default target: test
```

The two most interesting targets are the `test` and `test.coverage` targets, both of which run the unit tests. They compile and weave the TJHD code with both the JHD and AJHD sources in separate locations. Then they run the `org.jhotdraw.TestAll` unit tests on both implementations, outputting the test result to two files: an xml file that will be for displaying the test results in html later, and another file that can be used for comparing the two test results. After that they run the TestCompliance suite, inputting the formerly generated files for comparing the test results. The TestCompliance suite result is only outputted in xml format. Finally they generate test reports, in the form of a set of html pages that summarizes all test results. The `test.coverage` target creates two additional sets of html pages in two other directories, containing the code coverage information over JHD and AJHD. Figure 5.3 gives an impression of the test results pages, while Figure 5.4 shows the presentation of the test coverage results.

Custom helper classes

To be able to make this construction work, we have had to provide:

[3] `ant.apache.org`

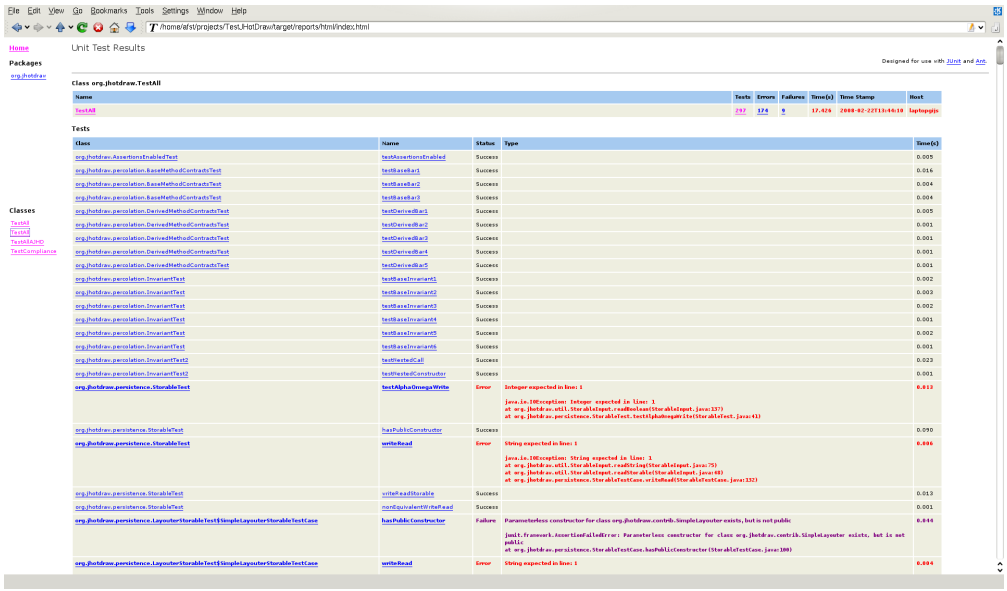

Figure 5.3: Screenshot of the test results html

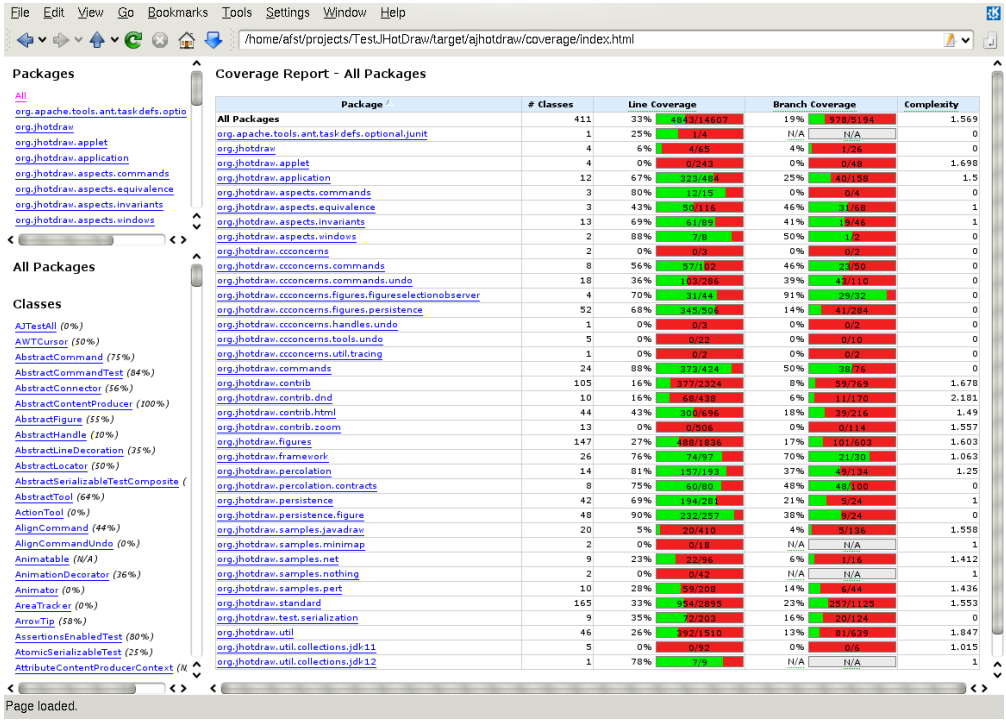

Figure 5.4: Screenshot of the test coverage html

- A custom ant xsl output file format to let ant display not only the testcase method names, but also the testcase class names.

- A custom test result class that can:

  - Extract the information needed for comparing test results. Unfortunately ant only recognizes atomic tests, so we could not display any test hierarchy information (composite tests).
  - Serialize and deserialize these results.
  - Compare the test result with another test results object, and generate messages specifying why the results are unequal

- A custom ant `JUnitResultFormatter` class to allow ant to store and retrieve these serializable test results.

- A custom junit `Description`, also functioning as a custom junit `Runner`. This class is internally used by the `TestCompliance` test run to simulate a junit test run, when actually the results of the two `TestAll` test runs are compared. It signals a test failure whenever one of test results are unequal, accompanied by the message generated by the test result class.

Coverage information

As coverage criteria for aspect-oriented programs are still in their infancy, the quality of the coverage information provided by cobertura, though better than nothing, is less then satisfying. Cobertura applies line, branch and condition coverage goals on both advice code and inter-type declarations. It does however not measure any coverage over pointcut expressions, nor has any capability of measuring coverage information relating to the application's join points. Cobertura also appears to suffer some bugs: it appears to attempt to measure the coverage over elements that can never be covered, such as attempting to achieve line coverage for `declare parents` statements. This and the large amount of bugs and sometimes nonoptimal design of JHD all contribute to the fact that a test coverage rate of 100% is often unachievable. These limitations aside, the information cobertura does provide can still indicate some missing test cases, and we did make use of it to some effect.

5.3 The persistence CCC

5.3.1 Implementation in JHD

The persistence concern deals with writing drawings to and retrieving them from persistent storage. The persistence concern is an instance of the role superimposition sort [31].

Figure 5.5 shows the key classes dealing with persistence. JHD provides four distinct ways of persisting drawings. These are represented by different StorageFormat classes.

Storable: This is the default way of persisting objects. Only objects that implement the `Storable` interface (listing 5.1) can be persisted. Persistence is handled by invoking the objects read and write method superimposed by the `Storable` interface.

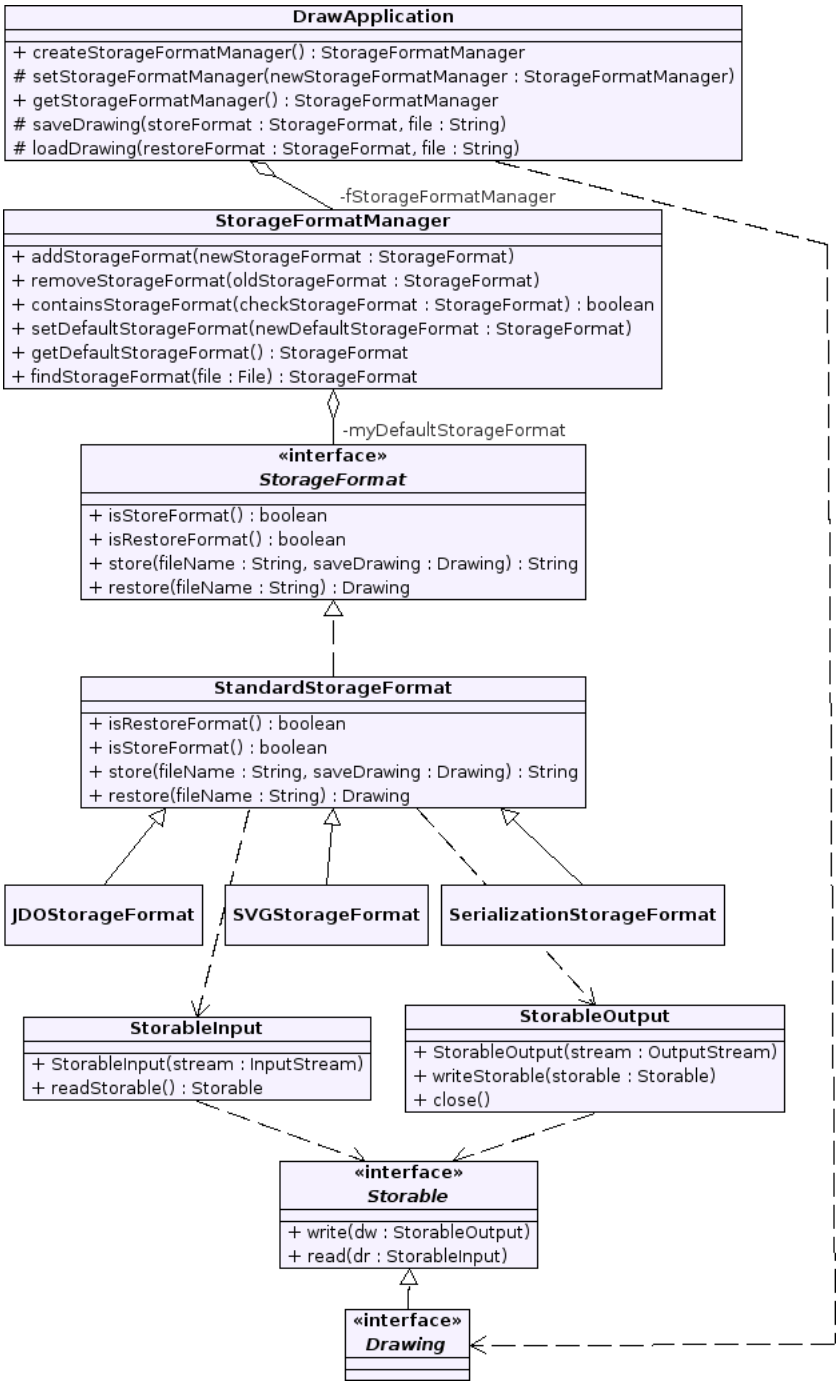

Figure 5.5: Key classes of the persistence concern in JHD

JDO: In this case, persistence is handled by the Java Data Objects (JDO) library by invoking several methods in the JDO `PersistenceManager`. This method does not require objects to implement a specific interface or extend a specific class, thus making it non-crosscutting. It remains unused, and has been removed in

```
1  /**
2   * Interface that is used by StorableInput and StorableOutput
3   * to flatten and resurrect objects. Objects that implement
4   * this interface and that are resurrected by StorableInput
5   * have to provide a default constructor with no arguments.
6   *
7   * @see StorableInput
8   * @see StorableOutput
9   *
10  * @version <$CURRENT_VERSION$>
11  */
12 public interface Storable {
13     /**
14      * Writes the object to the StorableOutput.
15      */
16     public void write(StorableOutput dw);
17
18     /**
19      * Reads the object from the StorableInput.
20      */
21     public void read(StorableInput dr) throws IOException;
22 }
```

Listing 5.1: The Storable interface

a later JHD version (although the maven pom file still contains the unused jdo dependency at the time of writing).

SVG: This is the only persistence method that only supports serialization (all other methods supports both serialization and deserialization). It stores a drawing in the Scalable Vector Graphics (SVG) format by invoking the `Drawing.paint` method with a special `Graphics` object which, instead of painting the drawing, stores the information necessary to save the image as an SVG file. This method of persistence is also not crosscutting.

Serialization: This persistence method uses the standard Java serialization framework. Like `Storable` serialization, every object has to implement an interface in order to be able to be persisted. The `java.io.Serializable` interface does not superimpose any methods and works without implementing any, but the default (de)serialization behavior can be overridden by implementing two specific *`private`* `static` methods, which the Java serialization mechanism is able to locate and invoke through reflection. It is also recommended to provide a `static SerialVersionUID` field to determine compatibility between the stored object and the class which is restoring it.

The different storage formats are being kept track of by the `DrawApplication`'s `StorageFormatManager`.

5.3.2 Aspect-oriented solution

The object-oriented solution contains two distinct crosscutting subaspects: the implementations of the `Storable` and `Serialization` interfaces. These features both

```
1  /**
2   * Adds the persistence concern to the
3   * Connector class and its subclasses.
4   */
5  public privileged aspect PersistentConnectors {
6      declare parents: Connector extends Storable;
7
8      /**
9       * Stores the connector and its owner to a StorableOutput.
10      */
11     public void AbstractConnector.write(StorableOutput dw) {
12         dw.writeStorable(owner());
13     }
14
15     /**
16      * Reads the connector and its owner from a StorableInput.
17      */
18     public void AbstractConnector.read(StorableInput dr)
19         throws IOException {
20         fOwner = (Figure) dr.readStorable();
21     }
```

Listing 5.2: Connector's Storable persistence unplugged

superimpose a secondary role on each class that can be persisted. Using AspectJ's language features, we can unplug this secondary role and put the implementation in a distinct package. This improves the class understandability, primarily because the code belonging to its core concern and secondary serialization concern can be viewed independently.

The Storable subaspect

We can easily unplug the Storable subaspect: a class that implements the `Storable` interface (listing 5.1) has to declare Storable its parent and implement all of the (public) methods superimposed by the interface. AspectJ has a `declare parent` construct that allows the parent declaration to be moved to an aspect. We can use inter-type method declarations to implement both `read` and `write` methods from the same aspect. Listing 5.2 contains an example (part of an) aspect which implements the Storable functionality for the Connector class and its direct descendant. Connectors in JHotDraw define connection points on a figure.

The Serialization subaspect

As we highlighted in section 5.3.1, serialization for `Serializable` classes are by default handled by the java built-in serialization mechanism. The read and write operations can be overridden by specifying methods with the following *exact* signatures:

- `private void writeObject( java.io.ObjectOutputStream out ) throws IOException`

- `private void readObject( java.io.ObjectInputStream in ) throws IOException, ClassNotFoundException`

A serializable class is automatically given a generated version number. When loading a class, Java will throw an `InvalidClassException` if the numbers do not match. Since these numbers are very sensitive to changes in the class (and any aspects that operate on it), it is generally recommended that a serializable object overrides this version number, which can be done by specifying an `ANY-ACCESS-MODIFIER static final long serialVersionUID` field.

The Serialization subaspect proves more difficult to unplug. We cannot use inter-type declarations to provide a class with private methods: AspectJ does define private inter-type methods, but the private access is restricted to the *aspect*, not to the class. Ceccato et al. have actually published this solution repeatedly [41, 7, 6], but *it does not work*. When attempting this construction, Java will fall back to its default serialization mechanism.

The `serialVersionUID` field can also not be unplugged. AspectJ should in theory be able to implement the `serialVersionUID` field as a `public static final long`, but AspectJ bug #52105[4] states that a `final static` field declaration results in a non-final field, which in turn causes the Java serialization implementation to ignore the intertype declared field and use a generated version number instead.

Summarizing, a combination of an obscure design of the Java serialization mechanism, AspectJ limitations and faults caused us to be unable to properly unplug any of the crosscutting features in the Serialization subaspect, and we have ceased our attempts to do so. We have only refactored the Storable implementation, so we will also focus our testing efforts on this subaspect.

5.3.3 Black-box tests

The responsibilities of the persistence concern are a bit ambiguous. As usual in JHD, documentation on the exact responsibilities is scarce to nonexistent, but most of it can be guessed. All conceivable drawings should be able to be stored and retrieved, and the stored drawing and its subsequently restored equivalent should be equal, the meaning of equal in this case depending on the method of storage. For the SVG storage format, we can only expect a visual equivalence. However, we have only refactored `Storable` persistence, so we will focus only on this subaspect. As we are dealing with a polymorphic class hierarchy, we apply Binder's Polymorphic Server Test Pattern [5, p. 513-517]. This test pattern targets inheritance-related faults resulting in incorrect of inconsistent responses in overridden methods. In short, it requires testing a superclass responsibilities on both the superclass itself (unless it is an interface) and all its subclasses, assuming LSP compliance [26].

The `Storable` class has some 60 subclasses, which we all provided with a suitable test class. Because of the large number of subclasses we have tried to write as much general code as possible to minimize the code needed for writing each specific test class.

[4] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=52105`

The `Storable` interface, as can be seen in listing 5.1, at least requires the implementing class to provide a default constructor with no arguments 'if they are resurrected by StorableInput'. We cannot easily discover if a Storable object can ever be resurrected by StorableInput, so in principle we test whether all Storable objects have a public no-argument constructor. While not explicitly specified in the `Storable` documentation, it does require the constructor to be public in order to access it. We do, of course, store and retrieve each implementing subclass at least once, and compare the resulting `Storable` object with the original one. Since JHD does not offer any equality checks for this, we have to add these ourselves. We define test object equality as follows: two objects are equal in respect to serialization if all non-transient field values are also equal. We also perform a deep equality check for these field values. This compare method is not purely a black-box, but it should be considered an addition to the JHD code that improves testability rather than an integral part of the test suite. We test both the storage and retrieval of one object, and the storage and retrieval of two unequal ones (if possible) to verify whether these are still unequal to each other and pairwise equal to the original objects after retrieval. In this final testcase we try to change as much object properties as possible to deviate the stored object a much as possible from the default object that would be created by invoking the no-argument constructor.

The `IOException` in the read method is undocumented, so we assume that any subclass that is eligible to throw it will add the preconditions to its documentation, and we will consider any thrown IOException a bug otherwise. Since we already call the store and retrieve methods, we do not need an extra check for this as a thrown IOException automatically results in a testcase failure.

The read and write methods fail to specify whether respectively `StorableInput` and `StorableOutput` are allowed to be null, but we silently assume that this should never be the case.

5.3.4 Testcases in the refactored solution

The refactoring consists only of a number of inter-type declarations; consulting our fault model from chapter 4 we find a number of possible faults:

- Fault b1: Wrong method signature in introduction
- Fault b2: Wrong class name in introduction
- Fault b3: Omitted parent interface
- Fault b4: Wrong scope

All of these potential faults have been covered by the black-box tests as expected. As they would all cause behavioral changes in storage and retrieval (fault b3 would even fail to compile), no additional testing is required.

5.3.5 Observations

While the persistence concern was an interesting introduction to the JHD codebase and the possibilities and limitations of AspectJ, it did not prove a very interesting example

in respect to testing aspect oriented-related faults. Creating the testcases did, however, demonstrate the usefullness of AspectJ for testing: we could implement a method for testing object equality outside of the core code. Since this method was primarily useful for making persistence testing easier (by improving the observability of the affected objects) it would have made little sense to actually implement it in the application.

5.4 The undo CCC

5.4.1 Implementation in JHD

JHD makes use of the Command pattern [18] for implementing the execution of mutations in a drawing. Each individual mutation is represented by a `Command` object. A `Command` object has a method which returns an `Undoable` object, which provides the implementation for the optional undo and redo operations. Each `DrawingEditor` has its own `UndoManager`, which maintains stacks of executed `Command` objects for undoing and redoing. Adding the `Undoable` to the `DrawingEditor`'s undo stack is the responsibility of the `UndoableCommand` command wrapper class. This is the only concrete class that directly implements the `Command` interface instead of subclassing `AbstractCommand`. Figure 5.6 shows the key classes of the undo concern. The `Tool` and `Handle` classes have a similar support for undo, but these merely call the factory methods from within the command classes.

5.4.2 Aspect-oriented solution

The undo concern crosscuts the core command concern in a number of places [28]:

1. The `AbstractCommand.myUndoableActivity` field.

2. The accessor(`getUndoActivity`) and mutator(`setUndoActivity`) methods for this field implemented by the same class, defined by the `Command` class.

3. The nested `UndoActivity` classes implemented by most of the concrete commands that support undo functionality.

4. The factory methods for these undo activities.

5. The references to the before enumerated elements from non-undo related members, e.g., the `execute()` method of the `Command` class.

These elements have been moved our of their command classes to separate aspect classes. Each refactored command now comes with its own undo aspect, which holds the `UndoActivity` for that class and makes sure this `UndoActivity` is properly constructed and set whenever the `execute()` method is called. The `getUndoActivity` and `setUndoActivity` methods have been removed from `Command` and defined in a new interface, `CommandUndoRole`. The `Command` interface extends `CommandUndoRole`. An example of a concrete command and its refactored equivalent is available in listings C.1, C.2 and C.3 on pages 99 to 104.

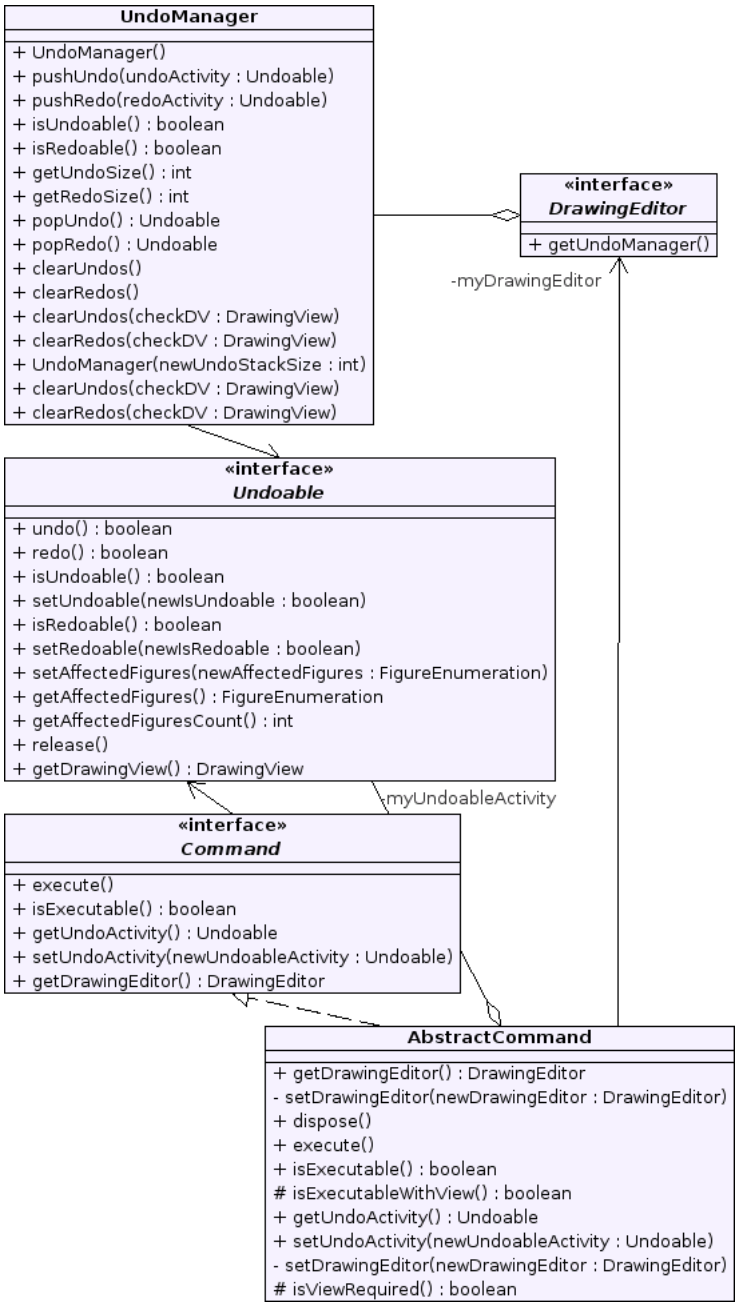

Figure 5.6: Key classes of the undo concern in JHD

5.4.3 Black-box tests

We start by looking at the responsibilities imposed by the `Command` and `Undoable` interfaces, shown in Listings C.4 on page 105 and C.5 on page 107. As we can see from the listings, most methods declared by these interfaces are not accompanied by any documentation. This makes finding their respective responsibilities a hard job.

Let's take a look at the `Command.execute()` method, for example. We know from

the description of the Command pattern that this method implements the `Command`'s action (the actual mutation of the drawing). The `Command` class does not provide any documentation on `execute()`, but reading the source code of `AbstractCommand` (the superclass for virtually every concrete `Command`), we find there are 3 definitions for when a command should be able to be executed:

1. The `execute()` method body checks whether the view is not null, and throws an exception otherwise.

2. Documentation of the `isExecutable()` method states that per default, a command is executable when at least one figure is selected in the currently activated view.

3. The actual checks in the implementation of the `isExecutable()` method are more sophisticated: when a view is required, it also requires it to be interactive. Otherwise, it calls the `isExecutableWithView()` method (which per default just returns true).

We guessed that the original intent was as follows: the `execute()` method should be able to be executed iff the `isExecutable()` method returns true, and individual commands may individually specify when they can or cannot be executed by overriding the `isExecutable()` and `isExecutableWithView()` methods, the function of the latter being somewhat vague.

As for the `Command` hierarchy, we are again dealing with a polymorphic class hierarchy, so like we did in the black-box tests for persistence we have again applied Binder's Polymorphic Server Test Pattern [5, p. 513-517].

As the `Command` objects also implement a state machine, we could also expect to be able to apply the N+ test strategy [5, p. 242-259]. The conceptual state diagram for `Command` objects is shown in Figure 5.7. This state diagram might not be considered complete, since there are some implicit guards that are not shown in either the diagram or in the documentation of the `isUndoable()` and `isRedoable()` methods: a command should not be undoable when other commands have been executed (and not undone) after the command itself. Likewise, it should not be redoable when commands in a lower position on the undo stack have been undone (and not redone). These requirements could also be considered part of the undo stack implementation, however if we look at the actual JHD implementation it seems that neither of these classes perform any checks of this sort: in fact the undo stack, as implemented by the `UndoManager` class, leaves this up to the client of the undo manager, which might be a class in an application that uses the JHD framework. As this requirement is not implemented in the `Command` class, we have only implemented the test cases for the given diagram. Sneak path tests have been left out as there is no sneak path checking - that is, the code just behaves in some unspecified way if for example the `undo()` method is called twice successively.

We have created a `CommandTestCase` and `UndoableCommandTestCase` superclass respectively for not-undoable and undoable tests.

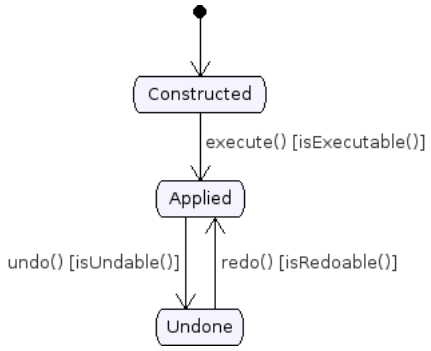

Figure 5.7: Command logical state diagram

5.4.4 Testcases in the refactored solution

A large part of the refactoring merely entailed moving members to inter-type declarations in aspects. These are already covered by the black-box test suite, and do not need additional aspect-oriented tests.

The aspects implementing the undo requirements for individual `Command` objects, however, do apply some pointcuts and advice to making sure the `UndoActivity` is properly set up when the `execute()` method is called. We will focus on one of those aspects, the `DeleteCommandUndo` aspect, as listed in listing C.3 on page 103.

From the pointcuts and advice faults, we have applied additional testcases for fault c1: Incorrect pointcut strength. Fault c2: Context exposure error also applies, but in this case we cannot easily check it. However, this should be exposed by the black-box tests: a context exposure error would cause the undo information to be stored in a different `Command` object, in which case undoing the original command should not work at all and undoing the other command should not work as expected. Fault d1: Wrong advice specification would be tested in a similar way, additionally requiring the advice to be run at least once at each join point, which the black-box also cover. Faults d3: Trying to modify arguments or return values in before or after advice, d2: Wrong or missing proceed in around advice and d4: Wrong or missing advice precedence do not require additional testcases.

We start out by applying our test for incorrect pointcut strength on the `commandExecuteInitUndo` pointcut, the only pointcut the `DeleteCommandUndo` aspect implements. Listing 5.3 lists pseudocode for the abstract pointcut invariant.

```
1 type: method execution (boolean) &&
2 declaring type: DeleteCommand or subclass (scalar) &&
3 method name: execute (boolean) &&
4 method return type: void (boolean) &&
5 method arguments: none (boolean)
```

Listing 5.3: Abstract class invariant pseudocode for the `commandExecuteInitUndo` pointcut in `DeleteCommand`

From this invariant we can extract 7 test cases, one of which is infeasible (an execute method with a non-void return type leads to a compile error). Interestingly, one of the testcases failed: the undo-related advice blocks did run when an `execute()` method

from a `DeleteCommand` subclass is executed. This could be considered a bug since the object-oriented does not (and could not) behave this way. While the tested command does not have any subclasses in JHD or AJHD, we cannot assume this to be the case in any third-party project that uses the JHD framework.

5.4.5 Observations

We found most aspect-oriented faults already to have been covered by the object-oriented testcases. Since the aspect-oriented code is merely a refactoring of an object-oriented equivalent, this result might partially be expected. However, re-checking the coverage of aspect-oriented bug hazards does provide better confidence in the correctness of the solution, and in this case we did discover some non-covered bug hazards, one of which even contained a (minor) bug. The boundary testing technique for pointcut strength we have applied here may be considered expensive to implement, but does seem to be effective.

We did encounter some difficulty concerning the testability of aspects: for one, privileged aspects are documented to be able to access all members including private ones, but this does not appear to hold for pointcuts. When an aspect under test (AUT) declares a pointcut private, it simply cannot be accessed by the test suite. In this case we can only check the pointcut strength by observing the behavior at a given join point, checking whether advice defined in the AUT runs or not. The same strategy should be applied for unnamed pointcuts.

5.5 Bugs found

Our black-box tests found a large number of bugs in JHD, and some AJHD-specific ones. Our final test set that runs on both JHD and AJHD contains 56 test failures. Both the AJHD-specific tests and the behavior preservation tests show a 100% success rate since all bugs these tests exposed were specific to AJHD, which we fixed upon discovery. Below we summarize both the bugs in JHD and AJHD we have discovered.

5.5.1 Bugs found in JHotDraw

- Some storable objects do not specify a public parameterless constructor, as required by the `Storable` interface.
- Several storable objects contain a buggy `read()` and/or `write()` methods.
- The `InsertImageCommand` class does not add the inserted image to affectedFigures upon execution.
- In a `JavaDrawApp`, undo does not work for an `InsertImageCommand`.
- For some commands (e.g., `DuplicateCommand`), undoing them fails to correctly restore the figure selection to its previous state
- Some commands handle multiple views in an `MDI_DrawApplication` (MDI stands for Multiple Document Interface) incorrectly. For example, when applying a `DeleteCommand` on a figure in one view, and subsequently undoing the

command when another view is active, the deleted figures will reappear in the latter view.

- Calling the `ImageFigure` constructor with a nullpointer image argument and a valid pathname throws a `NullPointerException` instead of loading the image. This is an interesting bug, since it seems to have been introduced by a bugfix for another bug. Better documentation or tests of the constructor would probably have prevented this.
- Across all `Command` subclasses, the `undo()` methods fail to restore the original image ordering. For instance, when we start our with drawing containing two figures (one behind the other), and we delete the figure in the back and subsequently undo the delete action, the image is reinserted in front of the other one. As apparent as it may seem, this is one of the bugs that have still persisted in the more recent JHD 7 at the time of writing.

5.5.2 Bugs found in AJHotdraw

- On several occasions, pointcuts defined in AJHD were defined in such a way that joinpoints in client code using the AJHD framework could possibly be affected. While in some cases this was intentional (mostly for `call()` pointcuts), in some cases it was not (e.g., for some `execution()` pointcuts).
- The `CommandContracts` aspect contained an `after()` advice, which should have been `after() returning` to represent the original control flow.
- The pointcut used by both advice blocks in the `CommandContracts` aspect captures all `execute()` method executions, including those resulting from `super()` calls.
- Some blocks of advice defined in the `UndoableCommand` aspect executed before the advice defined in the `CommandContracts` aspect. This was the result of two faults: `UndoableCommand` applied `call` pointcuts instead of `execution` pointcuts, and the aspect precedence had not been specified.

Finally, we encountered one additional bug in the existing test set: some of the existing tests assumed that after a command's `execute()` method finished, the selected figures would always equal the command's affected figures. This does not hold for some commands, e.g., for commands that remove figures (`DeleteCommand`, `CutCommand`, ...).

5.6 CCCs encountered

We have attempted to test as many types of CCCs as possible. Some of them were only partly or not at all implemented in AJHD, in which case we have attempted to expand or supply an implementation. In some of these cases we could not locate an instance of the CCC in JHD. Some CCCs are not supported by AspectJ (or Java for that matter). Policy enforcement, finally, cannot be tested as it only generates compiler warnings or errors. Table 5.1 on the next page summarizes which CCCs we

Name	Implemented	Tested	Where
Consistent behavior	No	Yes	CommandContracts
Contract enforcement	Yes	Yes	(TJHD) Base, Derived
	See next chapter		
Entangled roles	No	Yes	Undo
Redirection layer	No	Yes	UndoableCommand
Add variability	No	No	N/A
	Might have been applied as an alternative (maybe better, for it could leave the original Undoable class intact for client apps) refactoring for UndoableCommand (now as RL)		
Expose context	No	No	N/A
	We were not able to find an instance of this CCC in JHD		
Role superimposition	Yes	Yes	Storable subclasses
Support classes for role superimposition	No	Yes	Undo
Policy enforcement	Yes	No	persistence
	Cannot be tested		
Exception propagation	No	No	
	Refactoring does not pose additional bug hazards		
Declare throws clause	No	No	
	Not supported		
Design enforcement	No	No	
	Not supported		
Dynamic behavior enforcement	No	No	
	Not supported		

Table 5.1: Summary of encountered CCCs

have encountered. The 'name' column describes the name of the CCC following M. Marin's original naming [31]. The 'implemented' column means we have partly or wholly implemented an instance of the CCC. The 'tested' column indicates whether we have created testcases for an implementation of the CCC. The 'where' column contains an example class or classes which are part of the CCC. Finally, each CCC has a section in the bottom right that optionally contains some additional comments.

Chapter 6

Contract Enforcement

Contract enforcement is one of the common CCCs identified by M. Marin et al. [31]. In contrast with the CCCs we have treated in the last chapter, JHD does not employ exhibit the contract enforcement CCC. Also, it is a great deal more complicated. This is why we have dedicated a separate chapter for this feature.

We will start out by outlining the design by contract engineering principle, which the CE CCC attempts to enforce. We will also outline some requirements a contract enforcement mechanism should ideally support. Then we will continue with an object-oriented reference implementation, followed by several aspect-oriented solutions. We have accompanied all of the presented solutions with an analysis listing which of the requirements it fulfills. Finally we present our final solution and list some initial observations about the subject.

6.1 Design by contract

Design by contract (also called programming by contract) is a widely adopted method programming technique, described by B. Meyer [34]. It applies an analogy of real-world contracts to software engineering, emphasizing on *obligations* and *benefits* as generally occur in a real-world contract. Design by contract can be applied to a part of the software which can somehow be invoked, like a method. It distinguishes a *client* and a *supplier*. We will be talking exclusively about contracts imposed on a class (documented by the class `invariant` and by its methods pre- and postconditions).

6.1.1 Pre- and postcontitions

The supplier supplies the contract, which describes the benefits for the client given that the client fulfills its obligations. The clients obligations are are described in the form of *preconditions*, while its benefits are described in the form of *postconditions*. Both have to be described in the supplier's documentation.

For an example of using pre- and postconditions, let's take an implementation of a method that calculates the square root of a given number, following the signature `double sqrt(double number)`. A likely precondition would be the requirement for the provided number to be non-negative. The postcondition could be the promise that the return value represents the square root of the given number (or, as number representations are usually constrained to a limited number of bits, the number that is

closest to the actual square root). Now this postcondition can only be guaranteed if the client adheres to the precondition. The outcome of the `sqrt` method is undetermined when the precondition is not met, i.e., for negative input.

Exceptions can be considered part of the postcondition. Their occurrence generally describes some reason why the normal postcondition can't be met by the supplier even though the preconditions have been met by the client. However, their occurence can be expected (and usually can be recovered from), and should be described in the documentation. For this reason they can be considered part of the postcondition in terms of contract.

6.1.2 Invariants

In addition to pre- and postcontditions, B. Meyer also describes the use of class *invariants*. A class invariant imposed on a class specifies constraints each object of that class must satisfy both before and after each client operation. For example, a class with an explicit state attribute might include an invariant stating that the state attribute should not be null. If this class would contain a public method that updates the state, this method would be permitted to set the state to null temporarily, but the state should be set to a non-null value before the method returns.

6.1.3 Inheritance

The LSP describes a constraint on subclasses which generally requires every instance of a superclass to be logically substitutable with an instance of any of its subclasses, i.e., subclasses inherit their superclasses' observable behavior (method contracts). Upholding the LSP when subclassing a class with a contract invokes some constraints on the subclass. Generally speaking, a subclass should 'require no more, promise no less': when a subclass overrides a superclass method, it should only weaken the preconditions and strengthen the postconditions as to ensure compliance with the superclass contract. It also should not throw any exceptions unspecified by the superclass, or throw specified exceptions for unspecified conditions. As for using invariants in subclasses, B. Meyer suggests treating them as postconditions, only allowing strengthening of invariant constraints in subclasses.

Section 6.1.5 on the facing page shows some requirements for contract enforcement mechanisms that follow from these properties.

6.1.4 Enforcing the contract

Under ideal circumstances (if programmers would make no faults), the contract conditions should never be violated. Explicitly checking them does however provide an opportunity to diagnose errors early. B. Meyer suggests using assertions for this purpose, a concept that in Java is reflected through the `assert` statement [19]. Using assertions we can add the assertions, pre- and postconditions as boolean expressions and check these when necessary.

Pushing contract enforcement one step further, we could attempt to make it impossible for the programmer to strengthen preconditions or weaken postconditions for subclasses. The most common approach for this problem, which is also the method Meyer suggests, is to automatically combine the pre- and postconditions of a method with

those of its parents. The precondition of a method invocation then succeeds if either its own preconditions succeed, or one of its parents preconditions (thus a disjunction of all preconditions from the called method upwards to the upmost method definition). A similar approach can be used for postconditions, save for the fact that postconditions are combined as a conjunction: all postconditions from the called method upwards to the upmost method definition have to hold.

6.1.5 Contract enforcement requirements

We have listed a number of requirements that contract enforcement mechanisms for Java programs should ideally fulfill. This list results from the shortcomings we found in existing solutions. Note that the Eiffel[1] language, which B. Meyer has incepted as a showcase for applying design by contract and contract enforcement, does fulfill these requirements. The resulting list is as follows:

1. Preconditions can be weakened.
2. Postconditions can be strengthened.
3. Subclasses can reuse the code from their superclassses for weakening preconditions and strengthening postconditions, i.e. subclasses do not have to duplicate precondition/postcondition checking code from their superclasses.
4. When a subclass calls a method from its superclass (by using the `super` keyword), this (nested) method call will be checked against the *superclass* pre- and postconditions.
5. The class invariant will not be checked within an invariant call, e.g., when an invariant method calls an accessor method, this accessor method will not trigger another invariant check (which would call the accessor method again, etc.).
6. The class invariant will not be checked within a nested call, e.g. when a class has a public method1 which calls a public method2, the invariant will only be checked at the start and end of the outermost (method1) call.
7. The class invariant will not be checked within a nested constructor call for the same object, e.g. when a class has a public constructor1 which calls a public constructor2, the invariant will only be checked at the start and end of the outermost (constructor1) call. Note that this should only apply for constructors which are invoked upon the same object: if a constructor creates a second object (using the `new` construct, the invariant for the second object should be checked at the end of that call.

Next we will illustrate requirements 4 and 7.

6.1.6 `super()` calls

We can illustrate requirement 4 with our example code. If the `Derived1.bar` method is called with an argument of value 0 (a valid value for that method), it will call the

[1] `http://www.ecma-international.org/publications/standards/Ecma-367.htm`

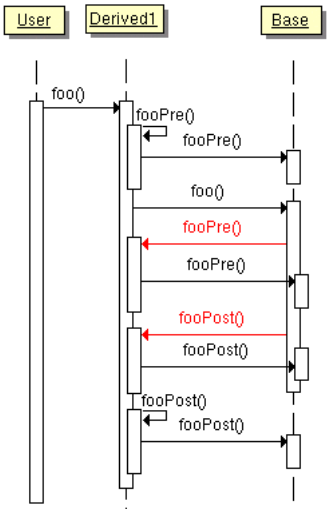

Figure 6.1: Sequence diagram for the `Derived.foo()` method from listing C.7 on page 110

`Base.bar` method with an argument of value 1 (also valid for the superclass). When the superclass method subsequently returns a (valid) 0 value, the postcondition assertion fails: the subclass postcondition is called instead of the superclass postcondition, and that postcondition disallows a 0 return value. Additionally, the `super.bar()` call would erroneously have succeeded if called with an invalid 0 value argument. Figure 6.1 shows the sequence diagram for the `Derived1.foo()` method. The `Base` object represents the superclass part of the `Derived1` object. The method calls marked in red are the illegal calls to the subclass pre- and postcondition enforcement methods.

6.1.7 Constructors

Requirement 7 on the previous page can be illustrated with the simple class hierarchy created by the code from listing 6.1. When we call the `Derived()` constructor, we would only expect the invariant to be checked just before our `new` statement returns. This corresponds with the two red lines in the sequence diagram shown in figure 6.2 on the next page. All of the other constructor returns result internal constructor calls which should not be checked, as the constructed object has not yet been fully initialized.

```
1  class Base {
2      protected Base(bool b) {
3      }
4
5      public Base() {
6          this(true);
7      }
8  }
9
10 class Derived extends Base {
11     protected Derived(bool b) {
```

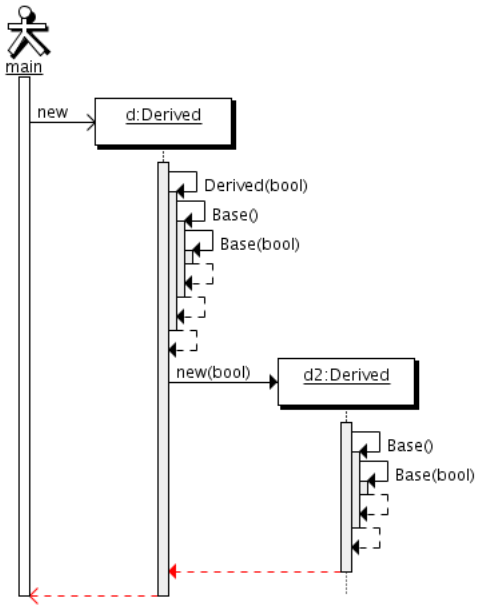

Figure 6.2: Sequence diagram for the `Derived()` constructor from listing 6.1 on the preceding page

```
12          super()
13      }
14
15      public Derived() {
16          this(b); // don't check invariant(s) yet.
17          new Derived(b); // do check invariant(s) here.
18      }
19
20      public static void main(String[] args) {
21          new Derived(); // only 2 invariant checks expected
22      }
23  }
```

Listing 6.1: A `Base` and `Derived` class with referencing constructors

6.2 Object-oriented implementation

Some object-oriented programming languages (including Eiffel, of course) offer native support for programming by contract. Java unfortunately does not[2], though some third-party solutions are available. We did not use any of these because we were attempting to find and test a cleanly moduralized AspectJ implementation. For this, we started out with an object-oriented reference implementation: Binder's Percolation pattern.

[2]Native support for programming by contract is listed, however, as the second most popular java enhancement requests at the time of writing (see `http://bugs.sun.com/top25_rfes.do`)

6.2.1 Percolation pattern

Binder's *Percolation pattern* [5, p.882-896] is an object-oriented solution for contract enforcement. This solution seems quite elegant and straightforward at first sight, but taking a second glance it will prove to exhibit typical faults which we encountered in several naive solutions for the CE problem. We have however been able to fix these problems, as we will show later on. Listings C.6 on page 109 and C.7 on page 110 illustrate a simple base class and a derived one implemented according to the percolation pattern.

Binder suggests adding a number of protected methods to each class with a contract: one `invariant()` method that returns true iff the invariant holds, and for each contracted method `xxx()` adding both an `xxxPre()` and an `xxxPost()` method that respectively return true iff the precondition and postcondition for the contracted method holds. Now all non-private methods can enforce the class contract by asserting their precondition before the method body, the postcondition after, and the invariant on both occasions. Constructors should assert the class invariant just before returning.

A subclass that overrides the method `xxx()` can now also override the `xxxPre()` and `xxxPost()` (and possibly `invariant()`) methods, reusing the contract code in the superclass by respectively calling the `super.xxxPre()`, `super.xxxPost()` and `super.invariant()` methods, as shown in the previously-mentioned listings C.6 on page 109 and C.7 on page 110. Preconditions should be combined by disjunction, postconditions and invariants by conjunction as to logically disallow precondition strengthening and postcondition or invariant weakening.

6.2.2 Faults in percolation

The Percolation pattern does not fulfill requirements 4 on page 67, 5 on page 67, 6 on page 67 and 7 on page 67 from our requirement listing:

- requirement 4 on page 67: Because the `super.xxxPre()` and `super.xxxPost()` methods are overridden, a `super()` call will also check the overridden (subclass) pre- and postconditions.
- requirements 5 on page 67, 6 on page 67 and 7 on page 67: The percolation pattern does not include any callstack checks, so the three callstack requirements do not hold.

6.2.3 Percolation improved

We can add support for `super()` calls by adding an extra method for both each pre- and postcondition. Listings C.8 on page 112 on page 112 and C.9 on page 115 on page 115show the improved versions of the `Base` and `Derived` classes from our previous example. The actual `xxxPre()` and `xxxPost()` methods have been made private, thus preventing themselves to be overridden. This way, a method never invokes a pre- or postcondition actually belonging to a subclass. To enable code reuse, we also add an additional protected `virtualXxxPre()` and `virtualXxxPost()` method, which in turn invoke the private `xxxPre()` and `xxxPost()` methods. These protected methods can then be called from the private pre- and postcondition methods to combine them with the subclasses own pre- and postcondition statements.

To add support for the callstack requirements for invariant checking (5 on page 67, 6 on page 67 and 7 on page 67), we have added a boolean attribute which, if turned off, disables invariant checks. Every public method then starts out by storing the current value of that attribute, then turns it off, and restores the value just before each method exit point. This construction covers requirements 5 on page 67and 6 on page 67. The last requirement (7 on page 67) is more difficult: since we cannot dynamically determine at the end of a constructor call whether that constructor is the result of a `this(..)` or `super(..)` call, we have to add separate constructors that can be told not check the invariant , so a subclass can call those constructors to postpone checking the invariant to the end of the outermost constructor execution.

One major downside of this solution is that it is quite error-prone, since an uncaught exception might bypass the code that re-enables invariant checking, in which case all future invariant checks would be disabled. The Java language is however not capable of providing enough information on the callstack to enable a more dynamic implementation of this feature. We can see that this problem already occurs whenever a methods precondition fails. We do not believe, however, that this is a bug in our program, as it results in an `AssertionError` which, since it is an `Error` and not an `Exception`, should not be caught as 'The class `Error` and its subclasses are exceptions from which ordinary pro- grams are not ordinarily expected to recover.' [19, p.306].

6.3 Aspect-oriented solution

The object-oriented implementation requires a lot of contract code to be added to a class, which we would hope to be able to remove using aspects. The AJHD project contained minimal contract enforcement code, only for invariant checking. As we wanted to test a more complete contract enforcement example, we have looked at the available solutions.

6.3.1 AspectJ team solution

As the AspectJ programming guide already emphasizes the usefulness of AspectJ for implementing contract enforcement[3], we were eager to apply this to our case study.

```
1  aspect PointBoundsChecking {
2
3      pointcut setX(int x):
4          (call(void FigureElement.setXY(int, int)) && args(x, *))
5          || (call(void Point.setX(int)) && args(x));
6
7      pointcut setY(int y):
8          (call(void FigureElement.setXY(int, int)) && args(*, y))
9          || (call(void Point.setY(int)) && args(y));
10
11     before(int x): setX(x) {
12         if ( x < MIN_X || x > MAX_X )
13             throw new IllegalArgumentException("x is out of bounds.");
14     }
15
```

[3] http://www.eclipse.org/aspectj/doc/released/progguide/starting-development.html#pre-and-post-conditions

```
16      before(int y): setY(y) {
17          if ( y < MIN_Y || y > MAX_Y )
18              throw new IllegalArgumentException("y is out of bounds.");
19      }
20  }
```

Listing 6.2: Contract enforcement example, as envisioned by the AspectJ team

Listing 6.2 on the preceding page shows the example from the AspectJ programming guide. This solution however has very bad support for inheritance: it fails to adhere to requirements 1 on page 67, 3 on page 67, 4 on page 67, 5 on page 67, 6 on page 67 and 7 on page 67. Both the pre- and postcondition clauses are implemented by before and after advice, which always runs on the affected method and all overridden ones. This means that preconditions can never be weakened, as we would need to override the before advice to do this. Advice can, however, not be overridden. Support for invariants are not even discussed in this solution. Finally the use of `call` pointcuts causes this code to be unsuitable for library classes unless the aspect is recompiled with the client code.

6.3.2 Solution by L. Briand et al.

Fortunately, L. Briand et al. [9] have proposed a solution with better support for inheritance, surpassing binder's percolation pattern by correctly dealing with `super()` calls. This solution, however, fails to support requirements 3 on page 67 (code reuse) and the callstack requirements for invariant checking (5 on page 67, 6 on page 67 and 7 on page 67). Also, it offers little improvements over the object-oriented code short of separating the contract code from the base code. And finally the invariant `after()` advice also runs after the invariant `before()` advice has already failed.

6.3.3 Our solution

As we could not find an aspect-oriented solution which fulfilled all of our requirements, we have come up with our own, integrating both code from Binder's percolation pattern and the solution proposed by Briand et al. The result can be found in listings C.10 on page 118 and C.11 on page 119 which show the example base classes, the contracts of which have been extracted to the aspects listed in listings C.12 on page 120 and C.13 on page 121. Invariant checking is handled by the aspect in listing C.15 on page 122, which checks the invariant from all classes implementing the interface given in listing C.14 on page 122.

We have been able to improve our object-oriented code- the `virtualXxxPre` and `virtualXxxPost` methods have become unnecessary as the inter-type declared pre- and postcondition methods can directly access private methods from their superclass. We can also represent Meyers original description for invariant checking more closely: Meyer originally stated that the invariant should be satisfied when a class instance is in an observable state [35, section 11.8, p.356]:

- On instance creation, that is after each constructor call.
- Before and after every remote call to a routine.

Solution	>pre	<post	pre+	post+	super	inv1	inv2	inv3
Percolation	+	+	+	+	-	-	-	-
Percolation imp.	+	+	+	+	+	+	+	+
AJ Team	-	+	N/A	+	-	N/A	N/A	N/A
Briand et al.	+	+	-	-	+	-	-	-
Our AO solution	+	+	+	+	+	+	+	+

Table 6.1: Properties for the given solutions for contract enforcement.
Legend:
Table headers:
>pre: ability to weaken preconditions in subclasses
<post: ability to strengthen postconditions in subclasses
pre+: precondition constraints can be reused in subclasses
post+: postcondition constraints can be reused in subclasses
super: pre/postconditions from superclass are checked for `super()` calls
inv1: invariant checking is disabled within `invariant()` calls
inv2: invariant checking is disabled for nested method calls
inv3: invariant checking is disabled for nested constructor calls
Table symbols:
+: supported
-: unsupported
N/A: not applicable

Unfortunately, AspectJ cannot bind arguments to in some negated pointcuts, so instead of using a simple `!this(w)` statement we had to use the more complicated `!if(thisJoinPoint.getThis() == w)` statement in the invariant aspect.

The non-crosscutting code (`xxxPre`, `xxxPost` and `invariant()` methods) can be moved back into the class if the developer likes to have the contract assertions close to the implementation, although this would require the re-addition of the `virtualXxxPre` and `virtualXxxPost` methods as presented in the object-oriented improved percolation pattern, as well as the declaring the `invariant()` protected or public. Pre- and postconditions for static methods, which are not included in our example, can simply be checked with basic before- and after advice.

6.3.4 Solutions overview

Table 6.3.4 summarizes the properties of the solutions we have described. As we can see, only our own solutions implement all of the requirements we have enumerated previously. We would, however, not recommend the object-oriented solution to be used in practice: it is quite error-prone, puts some tight implementation restrictions on the class and requires a lot of extra code. We believe the aspect-oriented solution to be usable: though the contracts from our sample code already outgrows the actual classes, these classes have a very minimal implementation - we would expect the complexity load of our solution to be relatively much smaller for a real-world class.

6.4 Testcases

Unfortunately we have only been able to apply our solution on a minor scale since JHD does not ship with any contract enforcement code. We have added a small number of classes to TJHD to be able to apply, and more importantly, test it. These classes have been especially designed for both good testability and diversity so we could both verify our claims regarding the behavior of our contract enforcement mechanism, and fine-tune our solution. Furthermore, we have tested the code in a similar fashion as we have done in the previous chapter.

6.5 Observations

- Implementing a contract enforcement mechanism that correctly deals with the LSP is not quite as trivial as it would appear on first sight. The faults in existing solutions illustrate this point.
- Our aspect-oriented solution still requires a bit of boilerplate code for each added pre- and postcondition. While AspectJ is designed to remove the necessity for this crosscuttingness, we could not find a way to do so in this particular case.
- Because of the `call` pointcut, any reusable classes applying our contract enforcement solution should be recompiled with all code that makes use of it. This constraint is a necessary evil, however, since for instance the invariant needs to be checked after the constructor call, which can be after the constructor execution of a client subclass.
- In our implementation we have not considered interactions between a class with a contract and other aspects. Other aspect could, for instance, change a private attribute or declare `around()` advice around the invariant advice, invalidating the class invariant without triggering an invariant check.

Chapter 7

Discussion

In this chapter we will discuss our results. We start out by evaluating the process we have followed in a chronological order, summarizing our decisions, general direction and sidetracks of our research. Then we will discuss and generalize our results by subject as far as this has not been covered by the process evaluation.

7.1 Process evaluation

7.1.1 Backwards engineering

When we fist started out with this project, the TJHD project only contained a small number of testcases, each of which could be applied to both JHD and AJHD. Running the testcases then required manual action to switch between testing the JHD and AJHD projects. One major disadvantage about JHD was the lack of testcases - for each refactoring we had to create a black-box test suite base first before starting to evaluating the specific aspect-oriented bug hazards. This process turned out to be a quite time-consuming, since the JHD code often lacked any documentation describing either method or class responsibilities, class contracts, package structure etc. As a result, we had to backwards-engineer most of the code responsibilities as much as this was at all possible. As most classes in JHD are part of some pattern, determining the classes general responsibilities generally was not very hard - most of our efforts went into determining precise method responsibilities.

7.1.2 Comparing test results

As we started implementing tests we soon had to rethink the test framework since we found that a number of newly added testcases were failing as a result of bugs in the JHD code. This made it hard to compare the JHD and AJHD test results. To address this problem we decided to add a test suite that could automatically compare the two former test suites. This also proved to be a nontrivial problem to solve, not in the least because the tests on JHD and AJHD needed to be compiled and run separately. We did however manage to get it done eventually.

7.1.3 CCC implementation

The third problem we encountered concerned the implemention of the CCCs identified by M. Marin [31]. For most, but not all of them, an implementation was included in the AJHD project. Since the description for most of these concerns and their aspect-oriented refactorings in particular were explained quite briefly, the possible range of their implementation structure would be quite large. This would make claims of the form 'This CCC has to be tested so-and-so' infeasible. We then decided to just test one or several instances of a particular CCC instead of trying to cover all of them.

Contract enforcement

One CCC in particular proved difficult to implement: the contract enforcement CCC. As we attempted to implement this CCC, something that we deemed quite easy beforehand, we discovered a large number of subtleties in its implementation, some of which had not even been accounted for in previously suggested solutions by others. In contrast to these solutions, however, we did manage to find an implementation which did deal with all of them.

7.1.4 Aspect-oriented testcases

As we had covered a number of CCCs we found that we were often testing the same language elements, and testing additional CCCs offered little additional value. We then moved our focus from the individual CCCs to the AspectJ language elements, determining per language element how it could be tested. We then reevaluated our testcases with the newly determined testing approaches, updated the testing approaches etc. At this time we also got a clearer image about possible adequacy criteria for aspect-oriented programs, which are closely related to grey-box testing techniques.

7.1.5 Achieved goals

When we originally we set out to find testing techniques for the various CCCs, we thought we would be able to describe a general approach for each individual CCC. Though we found out we would not be able to find a solution in this form, we did find out that by focusing on AspectJ language elements we could essentially deduce aspect-oriented test cases for arbitrary aspectj programs. We would argue that the grey-box tests we have devised currently represent the highest level of abstraction achievable; black-box tests would require additional modeling of aspect-oriented programs - as R. Binder tells us, "Test design cannot be accomplished without testable models. A good model is necessary to gain intellectual control over complex systems" [5, p.321].

We would also argue that since AJHD implements the same responsibilities as JHD and the black-box tests run on both the original implementations, we have covered the responsibilities of both implementations, and combined with the additional grey-box tests that cover a number of additional bug hazards specific to the aspect-oriented language constructions we have actually been able to achieve a pretty good coverage of the overall bug hazards in the refactorings.

The unfortunate lack of tool support for aspect-oriented adequacy criteria did make it harder to determine exactly how well our tests covered the implementation. The

aspect-oriented testcases did, however, find a number of bugs that the object-oriented testcases had missed, which does prove some amount of additional fault hazard coverage. Additionally we note that the grey-box test finding techniques closely resemble the adequacy criteria for aspects, leading us to assume that would we have had been able to measure the test adequacy better we would have achieved good code coverage.

7.2 Results evaluation

7.2.1 Fault model

Most faults from our fault model have been adopted from other fault models [43, 3, 4], though we did specify some of our own. We also left some of them out. Several reasons for doing this were because we did not think the faults were specific to aspect-oriented programs, because we did not think them to be pose a significant threat or because we thought them to be covered by other faults. As for the faults we adopted, we did often specify in more detail in what forms they could appear and in which ways they could influence the code correctness.

7.2.2 Coverage and adequacy criteria

Defining adequacy criteria was not one of our core goals, however our experience in testing aspect-oriented programs did lead us to some unanticipated insights in this matter, which we think would be a very interesting subject for further investigation. We have found that pointcuts can contain three distinct goals for which the expressions must be separated to be able to implement some interesting aspect-oriented adequacy criteria, arguing that by restricting the number of goals in dynamic pointcuts (including the addition of a more restricted `if`-like pointcut) it would be much easier to distinguish these goals. We have also proposed a new adequacy criterion, joinpoint execution coverage. Additionally we discussed a number of various concerns to take into account while defining and/or implementing a number of adequacy criteria.

7.2.3 Testing aspects

Pointcut testability

Aspects contained in a test suite cannot access all pointcuts from an IUT. While privileged aspects can access all members regardless of their visibility, including those declared as aspect members or by inter-type declarations, this does not hold for pointcuts: declaring an aspect privileged does not grant it additional visibility privileges for pointcuts. Unnamed pointcuts (pointcuts wholly or partly implemented in an advice specification) also cannot be accessed. This limits the testability of pointcuts: for non-accessible pointcuts we can only check their strength by observing the behavior of advice that is applied to these pointcuts in the IUT.

Advice testability

Unlike pointcuts, advice in AspectJ cannot be named at all. This makes it impossible for a test set to distinguish when a specific block of advice executes. While AspectJ

does supply an `adviceexecution()` pointcut, it can only supply the advice arguments and the aspect in which the advice is implemented, but when the aspect implements two blocks of advice with similar arguments, it is impossible to tell which one is actually executing. This means that as with testing pointcuts, in these cases we might have to rely on the behavioral properties of the advice to test them.

Pointcut strength

On several occasions we have seen that boundary testing the joinpoint selection strength of relatively simple pointcuts already requires a relatively large number of testcases. These did, however, uncover faults even in those simple pointcuts. We believe this evidence suggests that pointcuts are equally bug-prone as they are powerful, and the test effort we suggested is indeed necessary.

Our experience with testing pointcuts also seems to indicate that the most occurring type of faults in pointcuts are overgeneralizations in which the pointcut captures joinpoints that the core code does not yet contain, but could be added to the IUT in the future or by code outside of the IUT.

7.2.4 Additional AspectJ limitations

Advice inheritance

Once defined, advice cannot be extended or overridden. In some cases this can be especially burdensome, for instance when a method in some superclass applies a precondition check as implemented in the AspectJ programming guide [2], in which case a subclass cannot possibly weaken the precondition, because the advice from the superclass will throw an exception whenever the precondition from the superclass condition is not met.

Advice precedence

The AspectJ precedence system applies some intricate rules for determining precedence of different advice blocks within a single aspect. This is probably due to the difference in which the various types of advice are executed: blocks of `before()` and `after()` advice are executed in the order in which they are written, but two subsequent blocks of around advice A and B run in the order Aa-Ba-pointcut-Bb-Ab, where Xa denotes the part of the around advice code before the `proceed()` statement, and Xb the part after it. It would be easier to just reverse the order of `after` advice execution, this could eliminate the possibility of circular precedence altogether, and would not place additional (and counterintuitive) requirements on the implementation order between blocks of `before` and `after` advice. This change would also make it easier to replace `around` advice with `before` and `after` advice and vice versa, since it would not be necessary anymore to change the position of the advice in the aspect in order to perform this change without affecting the advice execution order.

Negating pointcuts with args

When implementing the contract enforcement we came across an in our opinion somewhat dubious restriction in AspectJ: some pointcuts, including `cflow` and `cflowbelow`,

cannot be negated if they include any arguments. The `cflow` and `cflowbelow` pointcuts cannot be used to bind any arguments (this can only be achieved with `this`, `target` or `args` pointcuts), so checking the control flow should only constitute a simple runtime check. While it is true an argument could possibly not be bound when the pointcut is negated, we would expect the problem still to be easily solvable: if it is possible to find all positive examples, all negative examples are simply all other ones (using a *closed world assumption* from formal logic).

In our implementation of the undo concern we have demonstrated an example usage of a negated `this` pointcut which could not be used for branching in this way, while the exact same function could actually be performed using an `if` pointcut. We believe this compiler limitation could actually be explained with our earlier mention of advice goals, and the apparent compiler inability to separate branching goals from context exposure goals in these pointcuts. Using arguments in negated pointcuts should be possible for pointcuts with only branching goals, but not for context exposure usage. We would argue this should even be possible to implement in an AspectJ compiler without changing the AspectJ language.

Chapter 8

Conclusions and Future Work

This chapter first gives an overview of our contributions. After this overview, we briefly summarize our conclusions. Finally, some ideas for future work will be discussed. For a more detailed overview and considerations we refer to chapter 7 on page 75: discussion.

8.1 Contributions

- We have identified three distinct pointcut goals. This separation proves a key concept for defining a number of adequacy criteria for aspect-oriented programs.
- We have introduced the 'join point execution coverage' adequacy criterion.
- We have presented a number of grey-box testing strategies for AspectJ programs, and applied them to a case study.
- We have implemented and tested most CCCs identified by M. Marin [31].
- We have created a testing framework for checking both responsibilities and behavior preservation between a Java program and its aspect-oriented refactored equivalent.
- We have presented a contract enforcement implementation in AspectJ which on several points surpasses previously suggested solutions.
- We have identified a number of possible improvements (or current shortcomings, if you will) for the AspectJ language.

All code written as part of this thesis are available via the SourceForge[1] open source repository.

8.2 Conclusions

In section 1.2 on page 2 we set out to *find out how to test common refactorings of crosscutting concerns in Java to AspectJ*. We are convinced the testing method we have

[1] `www.sourceforge.net`

applied adequately covers the most relevant bug hazards: black-box tests based on the unrefactored system still cover the code responsibilities in the refactored version, and the additional grey-box tests can be used to cover the additional bug hazards resulting from the new language elements.

8.3 Future work

Aspect-oriented adequacy criteria

Currently adequacy criteria for aspect-oriented programs for the most part only exist only on paper, and even most of them are still only vaguely defined. Researching these adequacy criteria, combined with providing code analysis tools capable of providing aspect-oriented coverage information based on these criteria could leverage testing techniques to a new level.

Responsibility-based tests for aspect-oriented programs

While we have focused on fault hazards occurring in specific language constructions, software designed using aspect-oriented models and design methodologies requires additional black-box tests tailored towards these models.

Improving testing approaches for aspect-oriented programs

Finding good testing approaches is an iterative process: they have to be applied, evaluated, refined and expanded continuously to achieve and maintain their effectiveness. Our testing approaches have only undergone a single iteration: we would recommend them to be reused in other projects and improved following newly acquired insights.

Application and further development of our contract enforcement solution

We have proposed an aspect-oriented contract enforcement mechanism. We would of course be very much interested to see it applied on a larger scale. it would also be interesting to research the possible interactions between classes with a contract and other aspects, and the implications this has for enforcing the contract.

Bibliography

[1] R.T. Alexander, J.M. Bieman, and A.A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Colorado State University, Mar 2004.

[2] The AspectJ Team. The AspectJ programming guide. `http://www.eclipse.org/{A}spect{J}/doc/released/progguide/index.html`, nov 2007.

[3] Jon S. Bækken and Roger T. Alexander. A candidate fault model for AspectJ pointcuts. In *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 169–178, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Jon S. Bækken and Roger T. Alexander. Towards a fault model for AspectJ programs: step 1 – pointcut faults. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 1–6, New York, NY, USA, 2006. ACM Press.

[5] Robert V. Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 2000.

[6] 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, Sept. 2006.

[7] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Grady Booch. *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley Publishing Company, 1996.

[9] L. C. Briand, W. J. Dzidek, and Y. Labiche. Instrumenting contracts with aspect-oriented programming to increase observability and support debugging. In IEEE Computer Society, editor, *21st IEEE International Conference on Software Maintenance (ICSM), Budapest, Hungary, September 25-30*, pages 687–690. IEEE, 2005.

[10] Gerardo Canfora and Luigi Cerulo. How crosscutting concerns evolve in JHotDraw. *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, 0, Sep 2005.

[11] Mariano Ceccato, Paolo Tonella, and Filippo Ricca. Is aop code easier or harder to test than oop code? In *On-line proceedings of the First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005)*, 2005.

[12] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley Professional, 2004.

[13] A. van Deursen, M. Marin, and L. Moonen. AJHotDraw: A showcase for refactoring to aspects. In *Proceedings of the Workshop on Linking Aspects and Evolution (LATE05)*, 2005.

[14] Arie van Deursen and Leon Moonen. The video store revisited - thoughts on refactoring and testing. In M. Marchesi and G. Succi, editors, *Proceedings of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*, May 2002.

[15] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[16] D. F. D'Souza and A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, Mass., 1 edition, 1999.

[17] Marting Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA, June 1999.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns*. Addison-Wesley, 1995.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[20] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, Reading, Massachusetts, June 1992.

[21] Ivar Jacobson, James Rumbaugh, and Grady Booch. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

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

[23] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[24] C. Koppen and M. Stoerzer. Pcdi: Attacking the fragile pointcut problem. In *First European Interactive Workshop on Aspects in Software (EIWAS)*, aug 2004.

[25] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *Proceedings of the Tenth ACM SIGSOFT Symposium on the Foundations of Softare Engineering (FSE-10)*. ACM Press, Nov 2002.

[26] Barbara Liskov. Data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, May 1988. ACM Press.

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

[28] Marius Marin. Refactoring JHotDraw's undo concern to AspectJ. In *Proc. of First Workshop on Aspect Reverse Engineering (WARE)*, 2004.

[29] Marius Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. PhD thesis, Delft University of Technology, 2008.

[30] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *Proceedings of the First International Workshop on the Modeling and Analysis of Concerns in Software, International Conference on Software Engineering*. St. Louis, USA, 2005.

[31] Marius Marin, Leon Moonen, and Arie van Deursen. A classification of crosscutting concerns. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*. IEEE Computer Society, 2005.

[32] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[33] Steve McConnell. *Professional Software Development*. Addison Wesley, June 2003.

[34] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[35] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1997.

[36] Gail C. Murphy, Paul Townsend, and Pok Sze Wong. Experiences with cluster and class testing. *Commun. ACM*, 37(9):39–47, 1994.

[37] Dewayne E. Perry and Gail E. Kaiser. Adequate testing and object-oriented programming. *Journal of Object-Oriented Programming*, 2(5):13–19, January–February 1990.

[38] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1):3, 2007.

[39] A. Rountev, A. Milanova, and B. Ryder. Fragment class analysis for testing of polymorphism in Java software, 2003.

[40] D. Shepherd and L. Pollock. Interfaces, aspects and views. In *Linking Aspect Technology and Evolution (LATE) Workshop*, 2005.

[41] Paolo Tonella and Mariano Ceccato. Migrating interface implementation to aspects. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 220–229, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Naoyasu Ubayashi and Tetsuo Tamai. Aspect-oriented programming with model checking. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154, New York, NY, USA, 2002. ACM Press.

[43] Arie van Deursen, Marius Marin, and Leon Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. Technical Report SEN-R0507, CWI, 2005.

[44] E. J. Weyuker. The evaluation of program-based software test data adequacy criteria. *Commun. ACM*, 31(6):668–675, 1988.

[45] Elaine J. Weyuker. On testing non-testable programs. *Computer Journal*, 25(4):465–470, November 1982.

[46] Dianxiang Xu and Weifeng Xu. State-based incremental testing of aspect-oriented programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 180–189. ACM, 2006.

[47] Dianxiang Xu, Weifeng Xu, and Kendall Nygard. A state-based approach to testing aspect-oriented programs. Technical report, Department of Computer Science, North Dakota State University, 2005.

[48] C. Zhang, H.-A. Jacobsen, J. Waterhouse, , and A. Colyer. Aspect refactoring verifier. In *1st Linking Aspect Technology and Evolution Workshop*, 2005.

[49] J. Zhao and M. Rinard. System dependence graph construction for aspect-oriented programs. Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.

[50] Jianjun Zhao. Data-flow-based unit testing of aspect-oriented programs. In *27th Annual International Computer Software and Applications Conference (COMPSAC)*. IEEE, 2003.

[51] Jianjun Zhao, Tao Xie, and Nan Li. Towards regression test selection for AspectJ programs. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 21–26, New York, NY, USA, 2006. ACM Press.

[52] Yuewei Zhou, Debra Richardson, and Hadar Ziv. Towards a practical approach to test aspect-oriented software. In *Proc. 2004 Workshop on Testing Component-Based Systems (TECOS 2004)*, sep 2004.

Appendix A

Glossary

This is an overview of frequently used terms and abbreviations. All terms that appear emphased (*like this*) are also individually listed.

Advice: A piece of executable code which can be applied to a number of *join points* described by a *pointcut*.

AJ: *AspectJ:* An aspect-oriented extension to the Java programming language.

AJHD: *AJHotDraw*

AJHotDraw: An aspect-oriented refactoring of the *JHotdraw* project.

AO: Aspect-Oriented

AOP: Aspect-Oriented Programming

AOSD: Aspect-Oriented Software Development

Aspect: Class-like aspect-oriented language element which can, besides methods and attributes, contain *inter-type declarations*, *pointcuts* and *advice*.

AspectJ: The reference aspect-oriented language for Java

AUT: *Aspect* Under Test

CCC: *Crosscutting concern*

CE: Contract Enforcement

Concern: A particular piece of interest or focus in a program.

Crosscutting concern: A *concern* which affects(crosscuts) other concerns

Du path: def-use path - the execution path starting at the definition of a variable to the use of it

Error: Human action producing a *fault*.

Failure: The manifested inability of a system to perform a required function.

Fault: Missing or incorrect code.

Inter-type declaration: The declaration of a type member or parent somewhere outside of the type itself (in an *aspect*).

Iff: If and only if.

ITD: *Inter-Type Declaration*

IUT: Implementation Under Test

JDO: Java Data Objects

JHD: *JHotDraw*

JHotDraw: A specific java project implementing a two-dimensional graphics framework.

JVM: Java Virtual Machine

join point: A point of execution in a (java) program.

LSP: Liskov Substitution Principle

OO: Object-Oriented

Pointcut: A collection of *join points*.

SVG: Scalable Vector Graphics

Appendix B

Bækken's fault model

The complete list of various pointcut faults devised by J. Bækken [3].

B.1 Faults in pointcuts

After each fault is listed which effects (infections) it could cause. The three symbols stand for whether they can respectively cause a PSE (Positive Selection Error) NSE (Negative Selection Error) and CEE (Context Exposure Error) effects.

Legend: $+$: Yes, $-$: No, $\otimes$: N/A

- Incorrect Pointcut Name — PSE NSE CEE
  - Method Call and Execution Pointcuts Mixed Up
    - Call should be execution — $++-$
    - Execution should be call — $++-$
  - Object Construction and Initialization Pointcuts Mixed Up
    - Call should be execution — $++-$
    - Execution should be call — $++-$
    - Initialization should be preinitialization — $++-$
    - Preinitialization should be initialization — $++-$
    - Call should be initialization — $++-$
    - Initialization should be call — $++-$
    - Execution should be initialization — $++-$
    - Initialization should be execution — $++-$
    - Call should be preinitialization — $++-$
    - Preinitialization should be call — $++-$
    - Execution should be preinitialization — $++-$
    - Preinitialization should be execution — $++-$
  - Cflow and Cflowbelow Pointcuts Mixed Up
    - Cflow should be cflowbelow — $+--$

    - Cflowbelow should be cflow −+−
  - This and Target Pointcuts Mixed Up
    - This should be target +++
    - Target should be this +++
  - Incorrect Name of User-Defined Pointcut
    - Incorrect name +++

- Incorrect Pointcut Argument(s)
  - Incorrect Method Pattern
    - Modifier pattern includes abstract together with static, final or synchronized −+−
    - Modifier pattern includes transient −+−
    - Modifier pattern includes volatile −+−
    - Declaring type pattern constitutes primitive type(s) −+−
    - Throws pattern does not constitute exception type(s) −+−
  - Incorrect Constructor Pattern
    - Modifier pattern includes abstract −+−
    - Modifier pattern includes static −+−
    - Modifier pattern includes final −+−
    - Modifier pattern includes transient −+−
    - Modifier pattern includes volatile −+−
    - Declaring type pattern constitutes interface(s) only −+−
    - Declaring type pattern constitutes primitive type(s) −+−
    - Attempting to use a method pattern to match constructors ++−
  - Incorrect Field Pattern
    - Modifier pattern includes abstract −+−
    - Modifier pattern includes synchronized −+−
    - Modifier pattern includes both final and volatile −+−
    - Declaring type pattern constitutes interface(s) only −+−
    - Declaring type pattern constitutes primitive type(s) −+−
  - Incorrect Type Pattern
    - Wildcard .. is used where * should be used ++−
    - Wildcard * is used where .. should be used ++−
    - Operator && is used between two types where || should be used −+−
    - Operator || is used between two types where && should be used +−−
    - Operator + is used after a type where it should not be used +−−
    - Operator + is not used after a type where it should be used −+−

    - Operator ! is used before a type where it should not be used $++-$
    - Operator ! is not used before a type where it should be used $++-$
    - Type is included that should not be included $++-$
    - Type is not included that should be included $++-$
    - Type(s) specified is (are) not visible in the scope of the pointcut $-+-$
    - Mutually exclusive types are &&-ed together $+--$
- Incorrect Modifier Pattern
    - Operator ! is used where it should not be used $++-$
    - Operator ! is not used before a modifier where it should be used $++-$
    - Modifier is included that should not be included $-+-$
    - Modifier is not included that should be included $+--$
    - Mutually exclusive modifiers are included $-+-$
    - Modifier inappropriate for the pattern is included $-+-$
- Incorrect Identifier Pattern
    - Wildcard * is used in a place where it should not be used $+--$
    - Wildcard * is not used in a place it should be used $-+-$
- Incorrect Parameter List Pattern
    - Incorrect number of parameters are listed $++-$
    - Order of parameters is incorrect $++-$
    - Parameter list pattern is empty when the wildcard .. should be used $-+-$
    - Wildcard .. is used where a type pattern should be used $+--$
- Incorrect Annotation Pattern
    - Operator ! is used before an element where it should not be used $++-$
    - Operator ! is not used before an element where it should be used $++-$
    - Element is included that should not be included $-+-$
    - Element is not included that should be included $+--$
- Incorrect Argument to This/Target Pointcuts
    - Wildcard * should be type $+--$
    - Wildcard * should be identifier $+--$
    - Type should be * $-+-$
    - Identifier should be wildcard * $-+-$
    - Type should be identifier $++-$
    - Identifier should be type $+++$
    - Type is incorrect type $++-$
    - Identifier is the incorrect pointcut/advice parameter $+++$

- Incorrect Arguments to Args Pointcut
  - Incorrect number of parameters is listed $+--$
  - Order of parameters is incorrect $+++$
  - Wildcard .. should be wildcard * $+--$
  - Wildcard * should be wildcard .. $-+-$
  - Wildcard .. should be type $+--$
  - Wildcard .. should be identifier $++-$
  - Type should be wildcard .. $-+-$
  - Identifier should be wildcard .. $-++$
  - Type should be identifier $++-$
  - Identifier should be type $+++$
  - Type is incorrect $++-$
  - Identifier is the incorrect pointcut/advice parameter $+++$
- Incorrect Argument to This/Target/Within/Withincode Annotation Pointcuts
  - Annotation type should be identifier $++-$
  - Identifier should be annotation type $+++$
  - Annotation type is incorrect $++-$
  - Identifier is the incorrect pointcut/advice parameter $+++$
- Incorrect Arguments to Args Annotation Pointcut
  - Incorrect number of parameters is listed $++-$
  - Order of parameters is incorrect $+++$
  - Wildcard .. should be wildcard * $+--$
  - Wildcard * should be wildcard .. $-+-$
  - Wildcard * should be annotation type $+--$
  - Wildcard * should be identifier $+--$
  - Annotation type should be wildcard * $-+-$
  - Identifier should be wildcard * $-++$
  - Wildcard .. should be annotation type $+--$
  - Wildcard .. should be identifier $+--$
  - Annotation type should be wildcard .. $-+-$
  - Identifier should be wildcard .. $-++$
- Incorrect Argument to Cflow/Cflowbelow Pointcuts
  - Incorrect pointcut expression $++-$
- Incorrect Argument to If Pointcut
  - Incorrect boolean expression $++-$
  - Expression has undesired side effect $\otimes\otimes\otimes$
  - Expression depends on side-effect of another pointcut $++-$
- Incorrect Arguments to User-Defined Pointcut

    - Order of parameters is incorrect ++−

- Incorrect Pointcut Composition
  - Incorrect or Missing Composition Operator
    - Operator || is used between two pointcuts where && should be used +−−
    - Operator && is used between two pointcuts where || should be used −+−
    - Operator ! is used before a pointcut where it should not be used ++−
    - Operator ! is not used before a pointcut where it should be used ++−
  - Inappropriate or Missing Pointcut Reference
    - Pointcut is referenced that should not be referenced ++−
    - Pointcut that should be referenced is not referenced ++−

B.2 Faults in advice

- Incorrect Advice Specification
  - Incorrect Type
    - Before should be after
    - After should be before
    - Before should be around
    - Around should be before
    - After should be around
    - Around should be afte
  - Incorrect Restriction of After Advice
    - After should be after returning
    - After returning should be after
    - After should be after throwing
    - After throwing should be after
    - After returning should be after throwing
    - After throwing should be after returning
  - Incorrect Returning/Throwing Parameter
    - Returning parameter is specified but should not be specified
    - Returning parameter is not specified but should be specified
    - Returning parameter has incorrect type
    - Throwing parameter is specified but should not be specified
    - Throwing parameter is not specified but should be specified
    - Throwing parameter has incorrect type

- Incorrect Advice Body
  - Missing or Incorrect Position of Proceed
    - Advice has a proceed statement but should not
    - Advice does not have a proceed statement but should
    - Proceed statement is guarded by condition it should not be guarded by
    - Proceed statement is not guarded by condition it should be guarded by
  - Incorrect Argument(s) to Proceed
    - Argument to proceed has incorrect value

Appendix C

Code listings

C.1 Summary

This appendix conatins the following listings:

- Listing C.1 on page 99, p. 99: The DeleteCommand class, before refactoring
- Listing C.2 on page 102, p. 102: The DeleteCommand class, after refactoring
- Listing C.3 on page 103, p. 103: The aspect implementing undo for DeleteCommand
- Listing C.4 on page 105, p. 105: The Command interface in JHD
- Listing C.5 on page 107, p. 107: The Undoable interface in JHD

These listings are referenced in section 5.4 on page 57, where we treat the undo CCC refactoring. The first three listings show an aspect-oriented refactoring of the undo CCC, whereas the last two listings show the interfaces we used whilst devising black-box tests for this CCC.

- Listing C.6 on page 109, p. 109: Percolation pattern, base class
- Listing C.7 on page 110, p. 110: Percolation pattern, derived class
- Listing C.8 on page 112, p. 112: Improved percolation pattern, base class
- Listing C.9 on page 115, p. 115: Improved percolation pattern, derived class
- Listing C.10 on page 118, p. 118: Aspect-oriented improved percolation pattern, base class
- Listing C.11 on page 119, p. 119: Aspect-oriented improved percolation pattern, derived class
- Listing C.12 on page 120, p. 120: Aspect-oriented improved percolation pattern, base class contract
- Listing C.13 on page 121, p. 121: Aspect-oriented improved percolation pattern, derived class contract

- Listing C.14 on page 122, p. 122: Aspect-oriented improved percolation pattern, invariant interface
- Listing C.15 on page 122, p. 122: Aspect-oriented improved percolation pattern, invariant aspect
- Listing C.16 on page 123, p. 123: Context-exposing aspect
- Listing C.17 on page 124, p. 124: Test cases for command contract

These are used in chapter 6 on page 65 to illustrate contract enforcement techniques. The first two listings apply Binder's percolation pattern [5, p.882-896]. Listings C.8 on page 112 and C.9 on page 115 show an object-oriented implementation containing our improvements of Binder's percolation pattern, and listings C.10 on page 118 through C.15 on page 122 show the aspect-oriented version. The final two listings show the (responsibility-based) testcases for the aspect-oriented solution.

C.2 Undo listings

Listing C.1: The DeleteCommand class, before refactoring

```
1  /*
2   * @(#)DeleteCommand.java
3   *
4   * Project:     JHotdraw - a GUI framework for technical drawings
5   *              http://www.jhotdraw.org
6   *              http://jhotdraw.sourceforge.net
7   * Copyright:     by the original author(s) and all contributors
8   * License:     Lesser GNU Public License (LGPL)
9   *              http://www.opensource.org/licenses/lgpl-license.html
10  */
11
12 package org.jhotdraw.standard;
13
14 import java.util.List;
15
16 import org.jhotdraw.framework.DrawingEditor;
17 import org.jhotdraw.framework.Figure;
18 import org.jhotdraw.framework.FigureEnumeration;
19 import org.jhotdraw.util.CollectionsFactory;
20 import org.jhotdraw.util.Undoable;
21 import org.jhotdraw.util.UndoableAdapter;
22
23 /**
24  * Command to delete the selection.
25  *
26  * @version <$CURRENT_VERSION$>
27  */
28 public class DeleteCommand extends FigureTransferCommand {
29
30     /**
31      * Constructs a delete command.
32      * @param name the command name
33      * @param newDrawingEditor the DrawingEditor which manages
34      * the views
35      */
36     public DeleteCommand
37         (String name, DrawingEditor newDrawingEditor) {
38         super(name, newDrawingEditor);
39     }
40
41     /**
42      * @see org.jhotdraw.util.Command#execute()
43      */
44     public void execute() {
45         super.execute();
46         setUndoActivity(createUndoActivity());
47         /* ricardo_padilha: bugfix for correct
48          * delete/undelete behavior
49          * When enumerating the affected figures we must not forget
50          * the dependent figures, since they are deleted as well!
51          */
52         FigureEnumeration fe = view().selection();
53         List affected = CollectionsFactory.current().createList();
```

```
54          Figure f;
55          FigureEnumeration dfe;
56          while (fe.hasNextFigure()) {
57              f = fe.nextFigure();
58              affected.add(0, f);
59              dfe = f.getDependendFigures();
60              if (dfe != null) {
61                  while (dfe.hasNextFigure()) {
62                      affected.add(0, dfe.nextFigure());
63                  }
64              }
65          }
66          fe = new FigureEnumerator(affected);
67          getUndoActivity().setAffectedFigures(fe);
68          /* ricardo_padilha: end of bugfix */
69          deleteFigures(getUndoActivity().getAffectedFigures());
70          view().checkDamage();
71      }
72
73      /**
74       * @see \
75       * org.jhotdraw.standard.AbstractCommand#isExecutableWithView()
76       */
77      protected boolean isExecutableWithView() {
78          return view().selectionCount() > 0;
79      }
80
81      /**
82       * Factory method for undo activity
83       * @return Undoable
84       */
85      protected Undoable createUndoActivity() {
86          return new DeleteCommand.UndoActivity(this);
87      }
88
89      public static class UndoActivity extends UndoableAdapter {
90
91          private FigureTransferCommand myCommand;
92
93          /**
94           * Constructor for <code>UndoActivity</code>.
95           * @param newCommand parent command
96           */
97          public UndoActivity(FigureTransferCommand newCommand) {
98              super(newCommand.view());
99              myCommand = newCommand;
100             setUndoable(true);
101             setRedoable(true);
102         }
103
104         /**
105          * @see org.jhotdraw.util.Undoable#undo()
106          */
107         public boolean undo() {
108             if (super.undo() &&
109                     getAffectedFigures().hasNextFigure()) {
110                 getDrawingView().clearSelection();
```

```
111                 setAffectedFigures(
112                     myCommand.insertFigures(
113                         getAffectedFiguresReversed(), 0, 0));
114                 return true;
115             }
116             return false;
117         }
118 
119         /**
120          * @see org.jhotdraw.util.Undoable#redo()
121          */
122         public boolean redo() {
123             // do not call execute directly as the selection
124             // might has changed
125             if (isRedoable()) {
126                 myCommand.deleteFigures(getAffectedFigures());
127                 getDrawingView().clearSelection();
128                 return true;
129             }
130             return false;
131         }
132     }
133 }
```

Listing C.1: The DeleteCommand class, before refactoring

Listing C.2: The DeleteCommand class, after refactoring

```
/*
 * @(#)DeleteCommand.java
 *
 * Project:     JHotdraw - a GUI framework for technical drawings
 *              http://www.jhotdraw.org
 *              http://jhotdraw.sourceforge.net
 * Copyright:       by the original author(s) and all contributors
 * License:     Lesser GNU Public License (LGPL)
 *              http://www.opensource.org/licenses/lgpl-license.html
 */

package org.jhotdraw.standard;

import org.jhotdraw.framework.DrawingEditor;
import org.jhotdraw.framework.FigureEnumeration;

/**
 * Command to delete the selection.
 *
 * @version <$CURRENT_VERSION$>
 *
 * @AJHD refactored: @author marin
 */
public class DeleteCommand extends FigureTransferCommand {

    /**
     * Constructs a delete command.
     * @param name the command name
     * @param newDrawingEditor the DrawingEditor which manages \
     * the views
     */
    public DeleteCommand
        (String name, DrawingEditor newDrawingEditor) {
        super(name, newDrawingEditor);
    }

    /**
     * @see org.jhotdraw.util.Command#execute()
     *
     * @AJHD refactored: consistent condition check
     * @see CommandContracts
     */
    public void execute() {
        FigureEnumeration fe = collectAffectedFigures();
    }

    /**
     * @see \
     * org.jhotdraw.standard.AbstractCommand#isExecutableWithView()
     */
    protected boolean isExecutableWithView() {
        return view().selectionCount() > 0;
    }
}
```

Listing C.2: The DeleteCommand class, after refactoring

Listing C.3: The aspect implementing undo for DeleteCommand

```
1  package org.jhotdraw.ccconcerns.commands.undo;
2
3  import org.jhotdraw.standard.DeleteCommand;
4  import org.jhotdraw.standard.FigureTransferCommand;
5  import org.jhotdraw.util.Undoable;
6  import org.jhotdraw.util.UndoableAdapter;
7
8  /**
9   * Undo support for DeleteCommand. Some of the more general concerns
10  * (ie, those that cover more Command elements) are in CommandUndo.
11  *
12  * @author Marius Marin
13  */
14 public privileged aspect DeleteCommandUndo {
15
16     /**
17      * Factory method for undo activity
18      * @return Undoable
19      */
20     /*@AJHD protected*/public Undoable
21         DeleteCommand.createUndoActivity() {
22         return new /*@ AJHD DeleteCommand*/
23             DeleteCommandUndo.UndoActivity(this);
24     }
25
26     //@see AlignCommandUndo
27     pointcut commandExecuteInitUndo(DeleteCommand acommand) :
28         this(acommand)
29         && execution(void DeleteCommand.execute());
30
31
32
33     //@see AlignCommandUndo
34     before(DeleteCommand acommand) :
35         commandExecuteInitUndo(acommand) {
36         acommand.setUndoActivity
37             (acommand.createUndoActivity());
38     }
39
40     //@see AlignCommandUndo
41     before(DeleteCommand acommand) :
42         commandExecuteInitUndo(acommand) {
43         acommand.getUndoActivity().setAffectedFigures
44             (acommand.view().selection());
45     }
46
47
48     //---------
49
50     public static class UndoActivity extends UndoableAdapter {
51
52         private FigureTransferCommand myCommand;
53
54         /**
55          * Constructor for <code>UndoActivity</code>.
56          * @param newCommand parent command
```

```
57            */
58           public UndoActivity(FigureTransferCommand newCommand) {
59               super(newCommand.view());
60               myCommand = newCommand;
61               setUndoable(true);
62               setRedoable(true);
63           }
64
65           /**
66            * @see org.jhotdraw.util.Undoable#undo()
67            */
68           public boolean undo() {
69               if (super.undo() &&
70                       getAffectedFigures().hasNextFigure()) {
71                   getDrawingView().clearSelection();
72                   setAffectedFigures(
73                       myCommand.insertFigures
74                       (getAffectedFiguresReversed(), 0, 0));
75                   return true;
76               }
77               return false;
78           }
79
80           /**
81            * @see org.jhotdraw.util.Undoable#redo()
82            */
83           public boolean redo() {
84               // do not call execute directly as the selection
85               // might has changed
86               if (isRedoable()) {
87                   myCommand.deleteFigures(getAffectedFigures());
88                   getDrawingView().clearSelection();
89                   return true;
90               }
91               return false;
92           }
93       }
94
95   }
```

Listing C.3: The aspect implementing undo for DeleteCommand

Listing C.4: The Command interface in JHD

```
1  /*
2   * @(#)Command.java
3   *
4   * Project:     JHotdraw - a GUI framework for technical drawings
5   *              http://www.jhotdraw.org
6   *              http://jhotdraw.sourceforge.net
7   * Copyright:     by the original author(s) and all contributors
8   * License:     Lesser GNU Public License (LGPL)
9   *              http://www.opensource.org/licenses/lgpl-license.html
10  */
11
12 package org.jhotdraw.util;
13
14 import org.jhotdraw.framework.DrawingEditor;
15
16 /**
17  * Commands encapsulate an action to be executed. Commands have
18  * a name and can be used in conjunction with <i>Command enabled</i>
19  * ui components.
20  * <hr>
21  * <b>Design Patterns</b><P>
22  * <img src="images/red-ball-small.gif" width=6 height=6 alt=" o ">
23  * <b><a href=../pattlets/sld010.htm>Command</a></b><br>
24  * Command is a simple instance of the command pattern without undo
25  * support.
26  * <hr>
27  *
28  * @see CommandButton
29  * @see CommandMenu
30  * @see CommandChoice
31  *
32  * @version <$CURRENT_VERSION$>
33  */
34 public interface Command {
35
36     /**
37      * Executes the command.
38      */
39     public void execute();
40
41     /**
42      * Tests if the command can be executed.
43      */
44     public boolean isExecutable();
45
46     /**
47      * Gets the command name.
48      */
49     public String name();
50
51     public DrawingEditor getDrawingEditor();
52
53     public Undoable getUndoActivity();
54
55     public void setUndoActivity(Undoable newUndoableActivity);
56
```

```
57      public void addCommandListener
58          (CommandListener newCommandListener);
59      public void removeCommandListener
60          (CommandListener oldCommandListener);
61  }
```

Listing C.4: The Command interface in JHD

Listing C.5: The Undoable interface in JHD

```
1  /**
2   * @(#)Undoable.java
3   *
4   * Project:     JHotdraw - a GUI framework for technical drawings
5   *              http://www.jhotdraw.org
6   *              http://jhotdraw.sourceforge.net
7   * Copyright:     by the original author(s) and all contributors
8   * License:     Lesser GNU Public License (LGPL)
9   *              http://www.opensource.org/licenses/lgpl-license.html
10  */
11
12 package org.jhotdraw.util;
13
14 import org.jhotdraw.framework.FigureEnumeration;
15 import org.jhotdraw.framework.DrawingView;
16
17 /**
18  * @author  Wolfram Kaiser <mrfloppy@sourceforge.net>
19  * @version <$CURRENT_VERSION$>
20  */
21 public interface Undoable {
22     /**
23      * Undo the activity
24      * @return true if the activity could be undone, false otherwise
25      */
26     public boolean undo();
27
28     /*
29      * Redo the activity
30      * @return true if the activity could be redone, false otherwise
31      */
32     public boolean redo();
33
34     public boolean isUndoable();
35
36     public void setUndoable(boolean newIsUndoable);
37
38     public boolean isRedoable();
39
40     public void setRedoable(boolean newIsRedoable);
41
42     /**
43      * Releases all resources related to an undoable activity
44      */
45     public void release();
46
47     public DrawingView getDrawingView();
48
49     public void setAffectedFigures
50         (FigureEnumeration newAffectedFigures);
51
52     public FigureEnumeration getAffectedFigures();
53
54     public int getAffectedFiguresCount();
55 }
```

Listing C.5: The Undoable interface in JHD

C.3 Contract enforcement listings

Listing C.6: Percolation pattern, base class

```
1  public class Base {
2      public Base() {
3          // constructor body
4          assert invariant();
5      }
6
7      /**
8       * A simple method.
9       */
10     public void foo() {
11         assert invariant();
12         assert fooPre();
13         // foo body
14         assert invariant();
15         assert fooPost();
16     }
17
18     /**
19      * A method with an actual pre- and postcondition.
20      * @pre arg>0
21      * @post rv>=0 (return value will never be negative)
22      */
23     public int bar(int arg) {
24         assert invariant();
25         assert barPre(arg);
26
27         int return_value = 0;
28
29         assert invariant();
30         assert barPost(return_value);
31         return return_value;
32     }
33
34     protected boolean invariant() { return true; }
35     protected boolean fooPre() { return true; }
36     protected boolean fooPost() { return true; }
37     protected boolean barPre(int arg) { return arg>0; }
38     protected boolean barPost(int rv) { return rv>=0; }
39
40 }
```

Listing C.6: Percolation pattern, base class

Listing C.7: Percolation pattern, derived class

```
1  public class Derived extends Base {
2      public Derived() {
3          // ctor body
4          assert invariant();
5      }
6
7      @Override public void foo() {
8          assert invariant();
9          assert fooPre();
10         // foo body
11         assert invariant();
12         assert fooPost();
13     }
14
15     /**
16      * A method with a weakened precondition
17      * and a strengthened postcondition.
18      * @pre arg>=0
19      * @post rv>0 (return value will always be positive)
20      */
21     @Override public int bar(int arg) {
22         assert invariant();
23         assert barPre(arg);
24
25         int return_value = super.bar(arg+1);
26         if(return_value==Integer.MAX_VALUE)
27             return_value = 1;
28         else
29             return_value+=1;
30
31         assert invariant();
32         assert barPost(return_value);
33         return return_value;
34     }
35
36     public void fum() {
37         assert invariant();
38         assert fumPre();
39         // fum body
40         assert invariant();
41         assert fumPost();
42     }
43
44     @Override protected boolean invariant() {
45         return true && super.invariant(); }
46
47     @Override protected boolean fooPre() {
48         return true || super.fooPre(); }
49
50     @Override protected boolean fooPost() {
51         return true && super.fooPost(); }
52
53     @Override protected boolean barPre(int arg) {
54         return arg==0 || super.barPre(arg); }
55
56     @Override protected boolean barPost(int rv) {
```

```
57          return rv!=0 && super.barPost(rv); }
58
59      protected boolean fumPre() { return true; }
60      protected boolean fumPost() { return true; }
61  }
```

Listing C.7: Percolation pattern, derived class

Listing C.8: Improved percolation pattern, base class

```
1  package percolation_oo;
2
3  public class Base {
4      private boolean checkInvariant = true;
5      private boolean invariantHolds = true;
6
7      public Base() {
8          this(true);
9      }
10
11     /**
12      * A constructor that can be used for testing invariant failures.
13      * @param invariantHolds whether the invariant should hold
14      *                       at ctor exit
15      */
16     public Base(boolean invariantHolds) {
17         this(invariantHolds, true);
18     }
19
20     /**
21      * Constructor that can be called by subclasses
22      * @param invariantHolds whether the invariant should hold
23      *                       at ctor exit
24      * @param checkInvariant whether to check the invariant
25      *                       at ctor exit
26      */
27     protected Base(boolean invariantHolds, boolean checkInvariant) {
28         this.invariantHolds = invariantHolds;
29         if(checkInvariant)
30             checkInvariant(true);
31     }
32
33     /**
34      * A method that can be used for testing invariant failures.
35      * @param invariantHolds whether the invariant should hold
36      *                       at method exit
37      * @param throwException whether the method should throw
38      *                       an exception
39      */
40     public void foo(boolean invariantHolds, boolean throwException) {
41         boolean inv = checkInvariant(false);
42         assert fooPre();
43
44         this.invariantHolds = invariantHolds;
45         if(throwException) {
46             checkInvariant(inv);
47             throw new RuntimeException("An exception occurred");
48         }
49
50         checkInvariant(inv);
51         assert fooPost();
52     }
53
54     protected boolean virtualFooPre() { return fooPre(); }
55     private boolean fooPre() { return true; }
56     protected boolean virtualFooPost() { return fooPost(); }
```

```
57      private boolean fooPost() { return true; }
58
59      /**
60       * A method that can be used for testing
61       * pre/postcondition failures.
62       * @pre arg>0
63       * @post rv>=0 (return value will never be negative)
64       */
65      public int bar(int arg) {
66          boolean inv = checkInvariant(false);
67          assert barPre(arg);
68
69          int rv = arg-2;
70
71          checkInvariant(inv);
72          assert barPost(rv);
73          return rv;
74      }
75
76      protected boolean virtualBarPre(int arg) { return barPre(arg); }
77      private boolean barPre(int arg) { return arg>0; }
78      protected boolean virtualBarPost(int rv) { return barPost(rv); }
79      private boolean barPost(int rv) { return rv>=0; }
80
81      /**
82       * Check the Base class invariant, unless both
83       * newCheck and checkInvariant are false.
84       * @arg newCheck the new value to set checkInvariant to
85       * @return the old value of checkInvariant
86       **/
87      protected boolean checkInvariant(boolean newCheck) {
88          boolean oldCheckInvariant = checkInvariant;
89          if(checkInvariant || newCheck) {
90              checkInvariant = false;
91              assert invariant();
92          }
93          checkInvariant = newCheck;
94          return oldCheckInvariant;
95      }
96
97      /**
98       * @return whether the base invariant currently holds
99       */
100     public boolean baseinvariantHolds() {
101         boolean inv = checkInvariant(false);
102         boolean ret = invariantHolds;
103         checkInvariant(inv);
104         return ret;
105     }
106
107     /**
108      * The actual invariant.
109      * Only to be called from the checkInvariant method.
110      **/
111     protected boolean invariant() {
112         return baseinvariantHolds();
113     }
```

```
114 }
```

Listing C.8: Improved percolation pattern, base class

Listing C.9: Improved percolation pattern, derived class

```
1  package percolation_oo;
2
3  public class Derived extends Base {
4      private boolean invariantHolds;
5
6      public Derived() {
7          this(true, true);
8      }
9
10     /**
11      * A constructor that can be used for testing
12      * invariant failures.
13      * @param baseInvariantHolds whether the base invariant
14      *                           should hold  at ctor exit
15      * @param derivedInvariantHolds whether the derived invariant
16      *                           should hold at ctor exit
17      */
18     public Derived
19         (boolean baseInvariantHolds, boolean derivedInvariantHolds) {
20         this(baseInvariantHolds, derivedInvariantHolds, true);
21     }
22
23     /**
24      * A constructor that can be called by subclasses.
25      * @param baseInvariantHolds whether the base invariant
26      *                           should hold at ctor exit
27      * @param derivedInvariantHolds whether the derived invariant
28      *                           should hold at ctor exit
29      * @param checkInvariant whether to check the invariant
30      *                           at ctor exit
31      */
32     protected Derived(boolean baseInvariantHolds,
33             boolean derivedInvariantHolds, boolean checkInvariant) {
34         super(baseInvariantHolds, false);
35         this.invariantHolds = derivedInvariantHolds;
36         if(checkInvariant)
37             checkInvariant(true);
38     }
39
40     /**
41      * {@inheritDoc}
42      */
43     @Override
44     public void foo(boolean invariantHolds, boolean throwException) {
45         boolean inv = checkInvariant(false);
46         assert fooPre();
47
48         super.foo(invariantHolds, throwException);
49
50         checkInvariant(inv);
51         assert fooPost();
52     }
53
54     @Override protected boolean virtualFooPre() {
55         return fooPre();
56     }
```

```
57
58      private boolean fooPre() {
59          return true || super.virtualFooPre();
60      }
61
62      @Override protected boolean virtualFooPost() {
63          return fooPost();
64      }
65
66      private boolean fooPost() {
67          return true && super.virtualFooPost();
68      }
69
70      /**
71       * Base.bar with a weakened precondition
72       * and a strengthened postcondition.
73       * @pre arg>=0
74       * @post rv>0 (return value will always be positive)
75       */
76      @Override public int bar(int arg) {
77          boolean inv = checkInvariant(false);
78          assert barPre(arg);
79
80          int rv = super.bar(arg);
81
82          checkInvariant(inv);
83          assert barPost(rv);
84          return rv;
85      }
86
87      @Override protected boolean virtualBarPre(int arg) {
88          return barPre(arg);
89      }
90
91      private boolean barPre(int arg) {
92          return arg==0 || super.virtualBarPre(arg);
93      }
94
95      @Override protected boolean virtualBarPost(int rv) {
96          return barPost(rv);
97      }
98
99      private boolean barPost(int rv) {
100         return rv!=0 && super.virtualBarPost(rv);
101     }
102
103     public void fum() {
104         boolean inv = checkInvariant(false);
105         assert fumPre();
106
107         // fum body
108
109         checkInvariant(inv);
110         assert fumPost();
111     }
112
113     protected boolean virtualFumPre() { return fumPre(); }
```

```
114     private boolean fumPre() { return true; }
115
116     protected boolean virtualFumPost() { return fumPost(); }
117     private boolean fumPost() { return true; }
118
119     @Override protected boolean invariant() {
120         return super.invariant() && invariantHolds;
121     }
122 }
```

Listing C.9: Improved percolation pattern, derived class

Listing C.10: Aspect-oriented improved percolation pattern, base class

```
1  public class Base {
2      public boolean baseInvariantHolds = true;
3
4      public Base() {
5          this(true);
6      }
7
8      /**
9       * Public method potentially triggering
10      * infinite invariant checking recursion.
11      * @return whether the invariant should hold.
12      */
13     public boolean baseInvariantHolds() {
14         return baseInvariantHolds;
15     }
16
17     /**
18      * A constructor that can be used for testing invariant failures.
19      * @param invariantHolds whether the invariant should hold
20      *                       at ctor exit
21      */
22     public Base(boolean invariantHolds) {
23         this.baseInvariantHolds = invariantHolds;
24     }
25
26     /**
27      * A method that can be used for testing invariant failures.
28      * @param invariantHolds whether the invariant should hold
29      *                       at method exit
30      * @param throwException whether the method should throw
31      *                       an exception
32      */
33     public void foo(boolean invariantHolds, boolean throwException) {
34         this.baseInvariantHolds = invariantHolds;
35         if(throwException) {
36             throw new RuntimeException("An exception occurred");
37         }
38     }
39
40     /**
41      * A method that can be used for testing
42      * pre/postcondition failures.
43      * @pre arg>0
44      * @post rv>=0 (return value will never be negative)
45      */
46     public int bar(int arg) {
47         return arg-2;
48     }
49 }
```

Listing C.10: Aspect-oriented improved percolation pattern, base class

Listing C.11: Aspect-oriented improved percolation pattern, derived class

```
1  package percolation;
2
3  public class Derived extends Base {
4      public boolean derivedInvariantHolds = true;
5
6      public Derived() {
7          // ctor body
8      }
9
10     /**
11      * {@inheritDoc}
12      */
13     @Override
14     public void foo(boolean invariantHolds, boolean throwException) {
15         super.foo(invariantHolds, throwException);
16     }
17
18
19     /**
20      * Base.bar with a weakened precondition
21      * and a strengthened postcondition.
22      * @pre arg>=0
23      * @post rv>0 (return value will always be positive)
24      */
25     @Override public int bar(int arg) {
26         return super.bar(arg);
27     }
28
29     public void fum() {
30         // fum body
31     }
32 }
```

Listing C.11: Aspect-oriented improved percolation pattern, derived class

Listing C.12: Aspect-oriented improved percolation pattern, base class contract

```
/**
 * @author Gijs Peek
 */
public privileged aspect BaseContract {
    declare parents: Base implements WithInvariant;
    // Base class contract (invariant)
    public boolean Base.invariant() {
        return invariantHolds();
    }

    // Base.foo contract
    private boolean Base.fooPre() {
        return true;
    }

    private boolean Base.fooPost() {
        return true;
    }

    void around(Base self): execution(void Base.foo(boolean, boolean))
        && target(self) && within(Base) {
        assert self.fooPre();
        proceed(self);
        assert self.fooPost();
    }

    // Base.bar contract
    private boolean Base.barPre(int arg) {
        return arg > 0;
    }

    private boolean Base.barPost(int rv) {
        return rv >= 0;
    }

    int around(Base self, int arg): execution(int Base.bar(int))
        && target(self) && within(Base) && args(arg) {
        assert self.barPre(arg);
        int result = proceed(self, arg);
        assert self.barPost(result);
        return result;
    }
}
```

Listing C.12: Aspect-oriented improved percolation pattern, base class contract

Listing C.13: Aspect-oriented improved percolation pattern, derived class contract

```
privileged aspect DerivedContract {
    public boolean Derived.invariant() {
        return super.invariant() && derivedInvariantHolds;
    }

    // Derived.foo contract
    private boolean Derived.fooPre() {
        return true || super.fooPre();
    }

    private boolean Derived.fooPost() {
        return true && super.fooPost();
    }

    void around(Derived self):
        execution(void Derived.foo(boolean, boolean))
        && target(self) && within(Derived) {
        assert self.fooPre();
        proceed(self);
        assert self.fooPost();
    }

    // Derived.bar contract
    private boolean Derived.barPre(int arg) {
        return arg==0 || super.barPre(arg);
    }

    private boolean Derived.barPost(int rv) {
        return rv!=0 && super.barPost(rv);
    }

    int around(Derived self, int arg): execution(int Base.bar(int))
        && target(self) && within(Derived) && args(arg) {
        assert self.barPre(arg);
        int result = proceed(self, arg);
        assert self.barPost(result);
        return result;
    }

    // Derived.fum contract
    private boolean Derived.fumPre() { return true; }
    private boolean Derived.fumPost() { return true; }
    void around(Derived self): execution(void Derived.fum())
        && target(self) && within(Derived) {
        assert self.fumPre();
        proceed(self);
        assert self.fumPost();
    }
}
```

Listing C.13: Aspect-oriented improved percolation pattern, derived class contract

Listing C.14: Aspect-oriented improved percolation pattern, invariant interface

```
/**
 * @author Gijs Peek
 */
public interface WithInvariant {
    public boolean invariant();
}
```

Listing C.14: Aspect-oriented improved percolation pattern, invariant interface

Listing C.15: Aspect-oriented improved percolation pattern, invariant aspect

```
package org.jhotdraw.percolation.contracts;

/**
 * This aspect automatically checks the invariant of all classes
 * that implement the WithInvariant interface.
 *
 * @author Gijs Peek
 */
public aspect InvariantAspect {
    // Check the invariant around each method call
    // originating from outside the object
    // except around calls to the invariant() method
    Object around(WithInvariant w):
        call(* WithInvariant+.*(..))
        && !call(public boolean WithInvariant+.invariant())
        && target(w) && !if(thisJoinPoint.getThis() == w) {
            assert w.invariant();
            try {
                return proceed(w);
            } finally {
                // also check invariant when exceptions are thrown
                // unfortunately, this might hide
                // the original error/exception
                assert w.invariant();
            }
    }

    // ... and after all constructor calls
    after() returning(WithInvariant w):
        call(WithInvariant+.new(..)) {
        assert w.invariant();
    }
}
```

Listing C.15: Aspect-oriented improved percolation pattern, invariant aspect

Listing C.16: Context-exposing aspect

```
1  package org.jhotdraw.ccconcerns.commands;
2
3  import org.jhotdraw.commands.CommandContractsTest;
4  import org.jhotdraw.standard.AbstractCommand;
5
6  /**
7   * This aspect exposes some CommandContracts context
8   * for the CommandContractsTest.
9   * @author Gijs Peek
10  */
11 public privileged aspect CommandContractsTestAspect {
12     declare precedence: CommandContracts, CommandContractsTestAspect;
13
14     private int hits;
15     private int hitsAtCallJoinpoint;
16
17     pointcut pc(AbstractCommand acommand) :
18         call(void AbstractCommand.execute())
19         && target(acommand)
20         && !if(acommand.getClass().isAnonymousClass());
21
22     before(AbstractCommand acommand):pc(acommand) {
23         hitsAtCallJoinpoint = hits;
24     }
25
26     after(AbstractCommand acommand) :
27         CommandContracts.commandExecuteCheckView(acommand) {
28         hits++;
29     }
30
31     public int CommandContractsTest.hits() {
32         return aspectOf().hits;
33     }
34
35     public int CommandContractsTest.hitsAtCallJoinpoint() {
36         return aspectOf().hitsAtCallJoinpoint;
37     }
38
39     public void CommandContractsTest.resetHits() {
40         aspectOf().hits = 0;
41     }
42 }
```

Listing C.16: Context-exposing aspect

Listing C.17: Test cases for command contract

```
1  package org.jhotdraw.commands;
2
3  import static org.junit.Assert.assertEquals;
4
5  import org.jhotdraw.application.DrawApplication;
6  import org.jhotdraw.framework.DrawingEditor;
7  import org.jhotdraw.framework.DrawingView;
8  import org.jhotdraw.framework.FigureSelectionListener;
9  import org.jhotdraw.standard.AbstractCommand;
10 import org.jhotdraw.util.Command;
11 import org.jhotdraw.util.CommandListener;
12 import org.jhotdraw.util.Undoable;
13 import org.junit.After;
14 import org.junit.Before;
15 import org.junit.Test;
16
17 /**
18  * Boundary test for the
19  * CommandContracts.commandExecuteCheckView pointcut.
20  * @author Gijs Peek
21  */
22 public class CommandContractsTest {
23     private DrawApplication testDApp;
24     private MyAbstractCommand cmd;
25
26     @Before
27     public void setUp() {
28         testDApp = new DrawApplication();
29         testDApp.open();
30         resetHits();
31         cmd = new MyAbstractCommand("", testDApp);
32     }
33
34     @After
35     public void tearDown() {
36         testDApp.exit();
37     }
38
39     @Test
40     public void testCommandExecuteCheckView1() {
41         // TC1: call joinpoint
42         cmd.execute();
43         assertEquals(0, hitsAtCallJoinpoint());
44         // sanity check: does the joinpoint actually work?
45         cmd.execute();
46         assertEquals(1, hitsAtCallJoinpoint());
47     }
48
49     // TC2: infeasible
50
51     @Test
52     public void testCommandExecuteCheckView3() {
53         // TC3: execute method with argument
54         cmd.execute(true);
55         assertEquals(0, hits());
56
```

```
57      }
58
59      @Test
60      public void testCommandExecuteCheckView4() {
61          // TC4: method not called execute
62          cmd.dispose();
63          assertEquals(0, hits());
64      }
65
66
67      @Test
68      public void testCommandExecuteCheckView5() {
69          // TC5: execute call on direct AbstractCommand subclass
70          cmd.execute();
71          assertEquals(1, hits());
72      }
73
74      @Test
75      public void testCommandExecuteCheckView6() {
76          // TC6: class implementing AbstractCommand's interfaces
77          new NoAbstractCommand().execute();
78          assertEquals(0, hits());
79      }
80
81      @Test
82      public void testCommandExecuteCheckView7() {
83          // TC7: Anonymous command
84          new AbstractCommand("", testDApp) {}.execute();
85          assertEquals(0, hits());
86      }
87
88      public class MyAbstractCommand extends AbstractCommand {
89          public MyAbstractCommand
90          (String newName, DrawingEditor newDrawingEditor) {
91              super(newName, newDrawingEditor);
92          }
93
94          public void execute(boolean foo) { /* stub */ }
95      }
96
97      public class NoAbstractCommand
98      implements Command, FigureSelectionListener {
99          public void execute() { /* stub */ }
100         public void addCommandListener
101             (CommandListener newCommandListener) { /* stub */ }
102         public DrawingEditor getDrawingEditor() { return testDApp; }
103         public Undoable getUndoActivity() { return null; }
104         public boolean isExecutable() { return true; }
105         public String name() { return ""; }
106         public void removeCommandListener
107             (CommandListener oldCommandListener) { /* stub */ }
108         public void setUndoActivity
109             (Undoable newUndoableActivity) { /* stub */ }
110         public void figureSelectionChanged
111             (DrawingView view) { /* stub */ }
112     }
113 }
```

Listing C.17: Test cases for command contract