A FAULT MODEL FOR POINTCUTS AND ADVICE IN ASPECTJ PROGRAMS

By

JON SWANE BAEKKEN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2006

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of JON SWANE BAEKKEN find it satisfactory and recommend that it be accepted.

____
Chair

____

____

ACKNOWLEDGEMENT

First of all I would like to thank my advisor and committee chair, Dr. Roger T. Alexander. I would also like to thank my two other committee members, Dr. David E. Bakken and Dr. Christopher D. Hundhausen. I would like to acknowledge the subscribers of *aspectj-users* who took their time to respond to my many (and sometimes stupid) questions to the mailing list; in particular Wes Isberg. Finally, I would like to thank my friends here in the States as well as home in Norway, for showing interest in my work and making sure that I found time for other activities in between.

PUBLICATIONS

Jon S. Baekken and Roger T. Alexander (to appear). A Candidate Fault Model for AspectJ Pointcuts. In *the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006), Raleigh, North Carolina*, November 6–10, 2006.

Jon S. Baekken and Roger T. Alexander. Towards a Fault Model for AspectJ Programs — Step 1: Pointcut Faults. In *the 2nd Workshop on Testing Aspect-Oriented Programs, Portland, Maine*, July 20, 2006, held in conjunction with the International Symposium on Software Testing and Analysis (ISSTA 2006).

A FAULT MODEL FOR POINTCUTS AND ADVICE

IN ASPECTJ PROGRAMS

Abstract

by Jon Swane Baekken, M.S.
Washington State University
August 2006

Chair: Roger T. Alexander

This thesis presents a fault model for pointcuts and advice, the two main constructs of the AspectJ programming language. The fault model provides a fault/failure analysis of how a fault, in a pointcut or a piece of advice in a program, can cause a data state in the program to become corrupted, and how that erroneous data state can propagate to the final state of the program, thereby manifesting a failure. The fault model also includes a catalog of fault types that are believed to represent faults likely to be introduced in programs by programmers writing AspectJ code. Each type of fault is described in terms of how it appears syntactically in source code as well as in how it can cause an infection of program state. The fault types are identified from a careful analysis of the syntax and the semantics of the pointcut and advice constructs. The fault model can help testers and programmers identify places in a program where faults are most likely to appear, and identify what kinds of faults to look out for when using a certain language feature. The fault model is claimed to be a good foundation for fault seeding, mutation testing, program inspections, and evaluation of testing strategies for AspectJ programs. Examples are given that demonstrate the model's suitability for these purposes. It is also believed that the fault model can be used to derive test adequacy criteria and devise testing strategies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Dedication

To my family,

for their continuous support.

CHAPTER ONE

INTRODUCTION

1.1 Aspect-Oriented Programming and AspectJ

Aspect-oriented programming (AOP) is an emerging programming paradigm that seeks new ways to modularize software systems. Modularizing involves separating and localizing the different concerns – things that we care about – in our system. Concerns can range from high-level requirements to low-level implementation issues. Due to limitations with the programming language, the implementation of a concern must sometimes be scattered across and/or tangled with the rest of the implementation. We say that such a concern is *cross-cutting*. While traditional programming paradigms have been successful in modularizing primary functionality ("core concerns") with constructs such as classes and functions, it has been argued that they fail to provide means of cleanly modularizing crosscutting concerns [28].

AOP attempts to solve the problem by allowing the programmer to develop cross-cutting concerns as full stand-alone modules called *aspects*. In most AOP languages, an aspect is comprised of one or more pieces of *advice* (code snippets - like methods) and a list of *join points* (points in the main program into which the advice should be woven). For example, a security module can include an advice that performs a security check, with instructions to *weave* this code snippet into the beginning of methods a(), b() and c() of some class. Powerful mechanisms enable a broad specification of join points, so that developers need not enumerate weaving-destinations manually. These mechanisms are commonly known as pointcut specification languages [6].

The term "aspect-oriented programming" was coined by Kiczales et al. at Xerox PARC in their widely cited 1997 paper bearing the same name [28]. Similar ideas as those presented by Kiczales et al. had however been published earlier by other researchers; most notably *subject-oriented* *programming* [25] (which later evolved into *multi-dimensional separation of concerns* [49]), *composition filters* [7] and *adaptive programming* [35]. The work at PARC eventually resulted in the first version of *AspectJ* in 1998 [34], the language that is now the most mature and widely-used aspect-oriented programming language [6]. The term aspect-oriented programming is now being used as a general term for the earlier projects mentioned, as well as for the new projects and languages continously appearing. A survey done by AOSD-Europe in 2005 listed 28 languages considered aspect-oriented in some way [16].

1.2 Software Testing and Fault Models

Software testing has a long history and has been defined in many ways. In his famous book from 1979, Myers [41] defined testing as “the process of executing a program with the intent of finding errors.” The IEEE Standard Glossary of Software Engineering Terminology [26] takes a more general approach and defines testing as “an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.” In other words, testing does not necessarily deal with the functional sides of a program, but could be an evaluation of performance or other non-functional aspects.

For functional testing, it is generally accepted that it comes in two main forms: *conformance-directed testing*, that seeks to establish conformance to requirements or specifications, and *fault-directed testing*, that seeks to reveal implementation faults [15]. The contribution of this thesis is in the area of fault-directed testing.

Binder [15] has used the term *fault model* to describe a model that “identifies relationships and components of a system under test that are most likely to have faults.” We cannot test all possible inputs, paths or states of a system, because there are simply too many of them. The cost would be tremendous for all but the simplest programs. In other words, exhaustive testing is not an option. Given this observation, we must have a way to design a test suite that exercises the program sufficiently to find most faults, yet is small enough to be practically useful. There will for sure be faults in any nontrivial software system — the question is where to look for them. A fault model answers this question for a specific programming paradigm or language.

1.3 Problem Statement

AspectJ [5] is an extension of the Java language that includes several new concepts and constructs. These include *join points*, which are well-defined points in the execution of a program, *pointcuts*, which are collections of join points, and *advice*, which are method-like constructs that can be attached to pointcuts, and thereby alter the program execution at the specified join points. *Aspects* are modular units comprising pointcuts and advice, besides ordinary Java member declarations. Because of all these new features, it is natural to assume that new kinds of faults can exist in programs written in the language.

We do not know what types of faults are unique to AspectJ programs, and this poses several restrictions on the development of programs in the language:

- Testers do not know where in a program faults are most likely to appear, and therefore do not know where to best direct the testing effort. In other words, testers lack a fundament for devising good testing strategies. Neither do testers know what impact various kinds of faults can have on the execution of a program. If a mistake is made when using a certain language feature, what can go wrong? What are the consequences?

- Programmers do not know what features of the language are likely to result in faulty code, and cannot take corresponding care when using those features. Because AspectJ is a relatively new language, most programmers have not yet acquired this knowledge through practical experience.

- Testers do not have test adequacy criteria that can tell them when a program has been adequately tested. Only when we have an idea of the nature of faults in a program can we set up rules for when the program has been adequately tested.

A fault model can help solve these problems. It will by itself identify places in a program faults are most likely to appear, as well as identify what kinds of faults a programmer should look out for when using a certain language feature. Further, a fault model can give insight into the possible consequences of the different kinds of faults, knowledge that can prove useful for example in debugging and localization of faults and test data selection.

A fault model will also provide a basis for further research on testing AspectJ programs. It can be used to devise test adequacy criteria, and can be a fundament for fault seeding [62] and mutation testing [20]. Once techniques for recognizing faults have been found, appropriate tools that can aid in testing can be identified. These tools could for example be tools for automatic generation of test cases or static analyzers looking for syntactic fault patterns in source code. A fault model could also be used to create checklists for code inspection. An AspectJ fault model could even be a starting point for creating a general fault model for aspect-oriented languages.

This thesis describe a fault model for pointcuts and advice, the two principal building blocks of aspects in the AspectJ language. The fault model is based on a careful analysis of the syntactic structures of pointcuts and advice, their semantics, and the possible impact they can have on program behavior.

1.4 Assumptions

In order to confine the scope of the thesis, some assumptions have been made:

- The fault model covers with faults resulting from *coding mistakes* rather than design mistakes or poor design decisions.

- Simple faults are covered, not complex faults. That is, faults in the fault model can be pinned down to one location in program code. Complex faults, in contrast, are faults that arise from several locations in program code having certain characteristics that collectively are incorrect.

- Faults appear one at a time. That is, interactions between faults, of the same type or of different types, are not considered.

- The fault model builds on the *competent programmer hypothesis* [19], which assumes that programs to be tested have been written by competent programmers, i.e., programmers "create programs that are close to being correct" [19]. Given this assumption, faults in a program should be detectable as small deviations from a correct program.

- Related to the above assumption, programmers are assumed be proficient with the AspectJ language. Faults that likely could be made by an intermediate to advanced programmer are targeted, not faults likely to be made only by beginners not yet familiar with the language.

CHAPTER TWO

BACKGROUND AND RELATED WORK

2.1 Background

This section introduces the AspectJ terminology[1] that will be used in the discussion of the fault model, and then gives an introduction the the concepts of faults and fault modeling.

2.1.1 The AspectJ Language

AspectJ [5] is an aspect-oriented extension of the Java language. A central concept in AspectJ is that of *join points*. A join point is any identifiable point of execution in a program, like a call to a method, the assignment to a variable or the execution of an exception handler. An *exposed* join point is a join point that is available for manipulation by an AspectJ program. A method call is an exposed join point, whereas the assignment to a local variable for example, is not. A *pointcut designator* is a construct that selects join points and sometimes *exposes context* at those join points. A pointcut designator can for example select all method calls to a specific method defined in a specific class and expose the method arguments. A *pointcut* can be a pointcut designator, or the actual set of join points selected by the designator. In the following we will use the term pointcut to mean a pointcut designator, unless stated otherwise. *Advice* is code to be executed at a join point that has been selected by a pointcut. Advice can executed before, after, or around the join point, and can make use of any context exposed by the pointcut. An *aspect* is a class-like modular unit that comprises pointcuts, advice, regular Java members, and *inter-type declarations*. Inter-type declarations are a way of introducing static changes to the classes, interfaces and aspects of a system, for example by specifying that a class should implement a specific interface.

[1]For the most part we follow the terminology used by the AspectJ Programming Guide [4], The AspectJ 5 Development Kit Developer's Notebook [2], and the AspectJ 5 Quick Reference [3], but we supplement with terminology from Laddad [30] and Colyer et al. [18]. The discussion in this thesis is based on AspectJ version 1.5.1.2006042612, AJDT version 1.4.0.2006042612 and Eclipse SDK version 3.2.0, build id I20060419-1640, using Java 2 Standard Edition version 1.5.0_06 under Mac OS X version 10.4.6.

Aspects and regular classes are combined by a process called *weaving*. Weaving usually occurs at compile-time but can also occur at at class load-time. Pointcuts can be regarded as *weaving rules* specifying how aspects are to be combined with classes (and other aspects). The weaving is performed by a tool called a *weaver*.

2.1.1.1 Join Points

A join point is a well-defined point in the execution of a program [4, Appendix B]. Only a subset of the join points in a program are available for AspectJ programs to reason about, and these are called *exposed join points* [30, page 43]. Because the exposed join points are usually the join points of interest, we will in the following use the term join point to mean an exposed join point, unless stated otherwise.

The kinds of (exposed) join points in AspectJ are [4, Appendix B]:

- **Method call:** when a method is called, not including super calls of non-static methods.
- **Method execution:** when the body of code for an actual method executes.
- **Constructor call:** when an object is built and that object's initial constructor is called (i.e., not *super* or *this* calls).
- **Constructor execution:** when the body of an actual constructor executes, after its *this* or *super* constructor call.
- **Static initializer execution:** when the static initializer for a class executes, i.e., the loading of a class.
- **Object pre-initialization:** before the object initialization code for a particular class runs. This encompasses the time between the start of its first called constructor and the start of its parent's constructor. In practical terms, this means calls made while forming arguments to the *super()* call in the constructor.

- **Object initialization:** when the object initialization code for a particular class runs. This encompasses the time between the return of its parent's constructor and the return of its first called constructor. It includes all the dynamic initializers and constructors used to create the object.

- **Field reference:** when a non-constant field is referenced.

- **Field assignment:** when a non-constant field is assigned a value.

- **Handler execution:** when an exception handler executes.

- **Advice execution:** when the body of code for a piece of advice executes.

Each join point potentially has three pieces of state associated with it: the currently executing object, the target object, and an object array of arguments [4, Appendix B]. We will in the following refer to these pieces of state as the *current object*, the *target object* and the *arguments* of a join point. The associated state for each kind of join point is listed in Table 2.1. Informally, the currently executing object is the object that the Java *this* pointer would refer to at a join point, and the target object is where control or attention is transferred to by the join point. The arguments are those values passed for that transfer of control or attention. Note that there is no executing object in static contexts such as static method bodies or static initializers, and there is no target object for join points associated with static methods or fields.

A join point can also have one or more Java metadata annotations associated with it. If an annotation has run-time retention the, value of the annotation can be accessed by the program at run-time [24]. Together, the state and annotations available at a join point constitute what in AspectJ is called the *context* at a join point.

An important property of a join point is its *signature* [2, Chapter 1]. Table 2.2 shows the constituent parts of of a join point signature for each kind of join point. A join point can have exactly one signature, except for the the method call, method execution, field reference and field

Join Point Kind	**Current Object**	**Target Object**	**Arguments**
Method call	Executing object	Target object	Method arguments
Method execution	Executing object	Executing object	Method arguments
Constructor call	Executing object	None	Constructor arguments
Constructor execution	Executing object	Executing object	Constructor arguments
Static initializer execution	None	None	None
Object pre-initialization	None	None	Constructor arguments
Object initialization	Executing object	Executing object	Constructor arguments
Field reference	Executing object	Target object	None
Field assignment	Executing object	Target object	Assigned value
Handler execution	Executing object	Executing object	Caught exception
Advice execution	Executing aspect	Executing aspect	Advice arguments

Table 2.1: Join points and their associated state

Join Point Kind	**Signature**
Method call	Return type, declaring type, identifier, id, parameter types
Method execution	Return type, declaring type, identifier, id, parameter types
Constructor call	Declaring type, parameter types
Constructor execution	Declaring type, parameter types
Static initialization	Declaring type
Object pre-initialization	Declaring type, parameter types
Object initialization	Declaring type, parameter types
Field reference	Declaring type, id, field type
Field assignment	Declaring type, id, field type
Handler	Exception types
Advice execution	Declaring type, parameter types

Table 2.2: Join point signatures

assignment join points, which can have multiple signatures, due to the fact that methods and fields can be inherited from super classes. Each signature of a method call or method execution join point has the same id and parameter types, but the declaring type and return type may vary. Each signature of a field reference or field assignment join point has the same id and field type, but the declaring type may vary.

Every join point has a single set of *modifiers* [2, Chapter 1]. These include the standard Java modifiers such as public, private, static, abstract etc., any annotations, and the throws clauses of

Join Point Kind	**Subject**
Method call	The method picked out by Java as the static target of the call
Method execution	The method that is executing
Constructor call	The constructor being called
Constructor execution	The constructor that is executing
Static initialization	The type being initialized
Object pre-initialization	The first constructor executing in this constructor chain
Object initialization	The first constructor executing in this constructor chain
Field reference	The field being referenced
Field assignment	The field being assigned
Handler	The declared type of the exception being handled
Advice execution	The advice being executed

Table 2.3: Join point subjects

methods and constructors. These modifiers are the modifiers of the *subject* of the join point. Table 2.3 shows the subject for each kind of join point.

Join point context, signatures and modifiers are all used by pointcuts for matching and/or exposure, as discussed in the next section.

2.1.1.2 Pointcuts

A pointcut can be defined with a pointcut declaration, which has one of the following forms [3]:

`abstract` [*Modifiers*] `pointcut` *Id* `(` [*Formals*] `);`

[*Modifiers*] `pointcut` *Id* `(` [*Formals*] `) :` *Pointcut expression*`;`

A pointcut can be defined in either a class or an aspect, and is treated as a member of that class or aspect. An aspect declared *abstract* can only be defined in an abstract aspect.

Modifiers can be *final*, and one of *private*, *protected* and *public*. *Id* is the name of the pointcut. *Formals* are parameters that can be bound to context at a join point by means of certain pointcuts. *Pointcut expression* can be any combination of simple and complex pointcuts, named and anonymous.

A *simple pointcut* is a pointcut that is not composed from other pointcuts and that does not have the *not* (!) operator before it. A *complex pointcut* is a pointcut composed with other pointcuts using the composition operators *and* (&&), and *or* (||), or that does have the *not* (!) operator before it.

Pointcuts can be either *named* or *anonymous*. A named pointcut is defined with the above declaration, and can be referred to by another pointcut. An anonymous pointcut has just the *Pointcut expression* part of the declaration, and thus has no name and cannot be referred to.

A pointcut *expression* is the part on the right-hand side of the pointcut (or advice) declaration, i.e., the part to the right of the colon (:). Sometimes the term “pointcut expression” is not used and just “pointcut” is used instead.

A *primitive pointcut* is a simple pointcut that comes built-in with the AspectJ language. Pointcuts that are not primitive are called *user-defined* pointcuts. There are three basic categories of primitive pointcuts [18, page 149], [2, Chapter 2]:

- **Pointcuts that match based on the kind of a join point (*kinded pointcuts*).** Each of these match one of the join point kinds discussed in the previous section. Pointcuts in this category are the *call*, *execution*, *initialization*, *preinitialization*, *staticinitialization*, *get*, *set*, *adviceexecution* and *handler* pointcuts.

- **Pointcuts that match based on join point context (*context pointcuts*).** These pointcuts match join points based on contextual information at a join point such as the values of arguments passed to a method or the presence of an annotation. They are also used to expose this context for other pointcuts or advice to use. Pointcuts in this category are the *this*, *target*, *args*, *if*, *@this*, *@target*, *@args*, *@within*, *@withincode* and *@annotation* pointcuts.

- **Pointcuts that match based on the scope in which a join point occurs (*scope pointcuts*).** These pointcuts match on the static or dynamic scope in which a join point occurs, such as inside a certain class, or in the control flow of a certain method. Pointcuts in this category are the *within*, *withincode*, *cflow* and *cflowbelow* pointcuts.

The general form and matching rules of the primitive pointcuts are [3, 2, 4]:

- **call** ($MethodPattern$). Selects each method call join point where *MethodPattern* matches at least one signature of the join point and the modifiers of the subject of the join point.
- **call** ($ConstructorPattern$). Selects each constructor call join point where *ConstructorPattern* matches the signature of the join point and the modifiers of the subject of the join point.
- **execution** ($MethodPattern$). Selects each method execution join point where *MethodPattern* matches at least one signature of the join point and the modifiers of the subject of the join point.
- **execution** ($ConstructorPattern$). Selects each constructor execution join point where *ConstructorPattern* matches the signature of the join point and the modifiers of the subject of the join point.
- **initialization** ($ConstructorPattern$). Selects each object initialization join point where *ConstructorPattern* matches the signature of the join point and the modifiers of the subject of the join point.
- **preinitialization** ($ConstructorPattern$). Selects each object pre-initialization join point where *ConstructorPattern* matches the signature of the join point and the modifiers of the subject of the join point.
- **staticinitialization** ($TypePattern$). Selects each static initialization join point where *TypePattern* matches the signature of the join point and the modifiers of the subject of the join point.
- **get** ($FieldPattern$). Selects each field reference join point where *FieldPattern* matches at least one signature of the join point and the modifiers of the subject of the join point.

- **set** ($FieldPattern$). Selects each field assignment join point where *FieldPattern* matches at least one signature of the join point and the modifiers of the subject of the join point.

- **handler** ($TypePattern$). Selects each handler execution join point where *TypePattern* matches the signature of the join point and the modifiers of the subject of the join point.

- **adviceexecution** (). Selects each advice execution join point.

- **within** ($TypePattern$). Selects each join point where the executing code is defined in a type whose signature and modifiers are matched by *TypePattern*.

- **withincode** ($MethodPattern$). Selects each join point where the executing code is defined in a method whose signature and modifiers are matched by *MethodPattern*.

- **withincode** ($ConstructorPattern$). Selects each join point where the executing code is defined in a constructor whose signature and modifiers are matched by *ConstructorPattern*.

- **cflow** ($Pointcut$). Selects each join point in the control flow of any join point P selected by *Pointcut*, including P itself.

- **cflowbelow** ($Pointcut$). Selects each join point in the control flow of any join point P selected by *Pointcut*, but not P itself.

- **if** ($BooleanExpression$). Selects each join point where *BooleanExpression* evaluates to $true$.

- **this** ($Type|Id|*$). Selects each join point where the *current object* (see Table 2.1) is an instance of *Type*, or of the type of the identifier *Id* (which must be bound in the enclosing pointcut or advice definition). The * wildcard stands for *any type*.

- **target** ($Type|Id|*$). Selects each join point where the *target object* (see Table 2.1) is an instance of *Type*, or of the type of the identifier *Id* (which must be bound in the enclosing pointcut or advice definition). The wildcard * stands for *any type*.

- **args** ($Type|Id| * |..,...$). Selects each join point where the *arguments* (see Table 2.1) are instances of the appropriate types (or of the identifiers if using that form). A null argument is matched iff the static type of the argument is the same as, or a subtype of, the specified *args* type. The * wildcard stands for any type, and the .. wildcard stands for *any number* of arguments of *any type*. There can be at most one .. wildcard for each *args* pointcut.

- **@this** ($Type|Id$). Selects each join point where the *current object* has an annotation of type *Type*, or of the type of the identifier *Id*. The annotation must have run-time retention.

- **@target** ($Type|Id$). Selects each join point where the *target object* has an annotation of type *Type*, or of the type of the identifier *Id*. The annotation must have run-time retention.

- **@args** ($Type|Id| * |..,...$). Selects each join point where the *arguments* have annotations of the appropriate types (or of the identifiers if using that form). The wildcard * stand for *any type*, and the .. wildcard stands for *any number* of arguments of *any type*. There can be at most one .. wildcard for each *@args* pointcut.

- **@within** ($Type|Id$). Selects each join point where the executing code is defined within a type that has an annotation of type *Type*, or of the type of the identifier *Id*.

- **@withincode** ($Type|Id$). Selects each join point where the executing code is defined withing a method or a constructor that has an annotation of type *Type*, or of the type of the identifier *Id*.

- **@annotation** ($Type|Id$). Selects each join point where the *subject* (see Table 2.3) has an annotation of type *Type*, or of the type of the identifier *Id*.

The pointcuts are explained in more detail in Chapter 4 in relation to the fault types associated with them.

As an example, consider the following pointcut that selects every call to a method named *someMethod* declared public, having any return type, number and types of parameters, where the target of the call is an instance of the class *SomeClass*, which is also exposed and bound to the formal parameter c:

```
pointcut somePointcut(SomeClass c) :
   call(public * someMethod(..)) && target(c);
```

2.1.1.3 Advice

A piece of advice is on the form [4, Appendix B]:

[`strictfp`] *AdviceSpec* [`throws` *TypeList*] `:` *Pointcut expression* { *Body* }

where *AdviceSpec* is one of

`before (` *Formals* `)`

`after (` *Formals* `) returning` [`(` *Formal* `)`]

`after (` *Formals* `) throwing` [`(` *Formal* `)`]

`after (` *Formals* `)`

Type `around (` *Formals* `)`

The purpose of an advice declaration is to attach behavior at each join point selected by a pointcut. *Pointcut expression* specifies this pointcut, which can be any combination of simple and complex pointcuts, named and anonymous.

Formals are parameters that can be bound to context at a join point by means of certain primitive pointcuts (see the previous section). The parameters can be used by the advice body to access this context.

There are three kinds of advice, and the kind determines how the advice interacts with the join points it is defined over. *Before* advice runs before its join points, *after* advice runs after its join points, and *around* advice runs both before and after (“around”) its join points, or in place of its join points. There are three interpretations of after advice: after the execution of a join point completes normally (*after returning*), after the join point returns with an exception (*after throwing*), or after the join point returns in either way (*after*). *Formal* for after returning can be used to expose the return value of the join point, whereas *Formal* for after throwing can be used to expose the thrown exception. The exposed return value and exception can be used by the advice body.

Around advice run both before and after or in place of the join points it is defined over. It must be declared with a return type compatible to the return type of the join point. The special syntax **proceed** ([*Arguments*]) can be used in the advice body to execute the original join point, and pass arguments from the advice to the join point. Without a proceed statement, the join point will not execute (and hence the advice will run in place of the join point)

The *strictfp* modifier has the same meaning as for a Java method (see [24]).

2.1.1.4 Aspect Example

This section presents a complete, but simple program that makes use of aspects, to show how pointcuts and advice work in practice. The example is taken from Laddad [30].

MessageCommunicator is a Java class that contains two methods that print messages; one to deliver a general message and the other to deliver a message to a specified person.

```
public class MessageCommunicator {
   public static void deliver(String message) {
      System.out.println(message);
   }

   public static void deliver(String person, String message) {
      System.out.print(person + ", " + message);
   }
}
```

Test is a simple class to test the functionality of *MessageCommunicator*.

```
public class Test {
   public static void main(String[] args) {
      MessageCommunicator.deliver("Wanna learn AspectJ");
      MessageCommunicator.deliver("Harry", "having fun?");
   }
}
```

The output from running the program is:

```
Wanna learn AspectJ?
Harry, having fun?
```

In Hindi, the suffix “ji” is often added to a person’s name to show respect. We now add an aspect to the program, that adds the suffix “ji” to the person’s name whenever a message is delivered to that person.

```
public aspect HindiSalutationAspect {
   pointcut sayToPerson(String person) :
      call(* MessageCommunicator.deliver(String, String))
      && args(person, String);

   void around(String person) : sayToPerson(person) {
      proceed(person + "ji");
   }
}
```

The output from the woven program is now:

```
Wanna learn AspectJ?
Harry-ji, having fun?
```

The example illustrates how the call to *deliver(String, String)* is captured by the pointcut, the first argument is exposed and bound to a formal parameter of the pointcut, and then to the formal parameter of the advice. Inside the body of the advice, when *proceed* is called, the original join point executes, which is the call to *deliver*, but now with the first argument of the call modified.

2.1.2 Fault Models

In the following, a *fault* will refer to a lexically incorrect statement in program source code. An *error* is an incorrect result following the evaluation of a statement. A *failure* is incorrect, observable output from the program.

Binder [15] uses the term *fault model* to describe a model that "identifies relationships and components of a system under test that are most likely to have faults." We cannot test all possible inputs, paths or states of a system, because there are simply too many of them. The cost would be tremendous for all but the simplest programs. In other words, exhaustive testing is not an option. Given this observation, we must have a way to design a test suite that exercises the program sufficiently to find most faults, yet is small enough to be practically useful. There will for sure be faults in any nontrivial software system — the question is where to look for them. A fault model answers this question for a specific programming paradigm or language. Binder identifies two general fault models and corresponding test strategies:

- **Conformance-directed testing**, that seeks to establish conformance to requirements. This type of testing relies on a *non-specific fault model*: any fault suffices to prevent conformance. A fault model for conformance-directed testing therefore need not consider potential implementation faults in detail, but must make sure that the test suite is sufficiently representable of the requirements of the system.

- **Fault-directed testing**, that seeks to reveal implementation faults. This type of testing requires a *specific fault model* to direct the search for faults, since the number of input, output, state and paths combinations is astronomical.

The focus of this thesis is on fault-directed testing, and when we write *testing* and *fault model* we mean *fault-directed testing* and *specific fault model* correspondingly, unless stated otherwise.

Most published research on software faults has not used the term "fault model", even if the same issues are discussed. For example, in [14], Binder used the term *fault hypothesis* for the same purpose:

> A fault hypothesis is an essential part of a testing approach. It is an assumption based on common sense, experience, suspicion, analysis or experiment about the relative likelihood of faults in some particular aspect of a system under test. A fault hypothesis answers a simple question about a test technique: 'Why do the features called out by the technique warrant our test effort?'

Binder also surveyed 41 different fault hypotheses for object-oriented programs, covering features such as inheritance, polymorphism and object state, and C++ specific faults. These works are not re-surveyed here, but can be found in [14].

Later work on this kind of fault models include Offutt et al. [43] and Alexander et al. [10], who created a fault model for subtype inheritance and polymorphism in object-oriented programs.

The term fault model is also used to describe any model or theory that explains something about the nature of software faults. Two such models that will be used in the AspectJ fault model are the *fault/failure model* [51] and the *RELAY model of error detection* [44].

The term *fault/failure model* is due to Voas, Morell and Miller [51] and Friedman and Voas [23], although DeMillo and Offutt [21] published similar work at about the same time (using other terminology). The theory behind the model goes back to Morell and Hamlet [39] and Morell [38].

The fault/failure model comprises three individually necessary and collectively sufficient conditions for a fault to produce a failure:

- The fault must be executed (*execution*).
- The succeeding data state must be infected (*infection*).
- The data-state error must propagate to the output (*propagation*).

In other words, the presence of faults in a program is no guarantee for a program failure. To understand why, we must consider the sequence of location executions that a program performs.

Each set of variable values after execution of a location in a computation is called a data state. After executing a fault, the resulting data state might be corrupted, in which case an *infection* has occurred and the data state contains an error. After the infection, the data state error has to propagate to the output, or a final observable state, of the program. At this point, we have a failure.

Consider the different possibilities we have when executing a program containing a fault [51]:

- The fault is not executed.
- The fault is executed but does not infect any data state.
- The fault is executed and some data states are infected, but the output is nonetheless correct.
- The fault is executed, infection occurs, and the infection causes an incorrect output.

Only the last possibility would make the fault visible to a tester; the other possibilities are cases of *coincidental correctness*.

Closely related to the fault/failure model is the RELAY model of error detection [44, 62], which also builds on the theory of Morell and Hamlet [39] and Morell [38]. In the RELAY model, a *potential fault* is a discrepancy between a node in the flow graph of the program under test and the corresponding node in the flow graph of a hypothetically correct program. This potential fault results in a *potential error* if the expression containing the fault is executed and evaluated to a value different from that of the corresponding hypothetically correct expression. Given a potential fault, a potential error *originates* at the smallest subexpression of the node containing the fault that evaluates incorrectly. The potential error transfers to a *superexpression* if the superexpression evaluates incorrectly. Such transfers are called *computational transfer*. To reveal an output error (i.e., a failure), execution of a potential fault must cause an error that transfers from node to node until an incorrect output results, where an error in the function computed by a node is called a *context error*. If a potential error is reflected in the value of some variable that is referenced at another node the error transfer is called a *data-flow transfer*.

The fault model described in Chapter 4 and Chapter 5 mainly builds on the fault/failure model as described in [51], but is also inspired by the RELAY model.

2.2 Related Work

This section provides a survey of related work. First, work on fault models for aspect-oriented programs is summarized, and then fault-directed testing strategies for aspect-oriented programs are surveyed.

2.2.1 Fault Models for Aspect-Oriented Programs

Alexander and Bieman [8] identified four alternatives that need to be considered when dealing with faults and failures in AOPs, arising from the assumption that an aspect cannot stand on its own — understanding an aspect requires knowledge of the base program it is woven into:

- **The fault resides in a portion of the base program that is not affected by a woven aspect.** The fault is unaffected by the data and control dependences induced by the woven aspect. Thus, the fault is peculiar to the base program and could occur if there was no weaving.

- **The fault resides in code that is specific to the aspect, isolated from the woven context.** In this case, the fault would be present in any composition that included the aspect. However, the fault resides in aspect code that is independent of the data and control dependences induced by the weaving process.

- **The fault is an emergent property that results from some interaction between the aspect and the base program.** This would occur when the result of the weaving process introduces additional data or control dependences not present in the base program or the aspect alone. Instead, these dependences arise from the integration and interaction of code and data between the base program and the aspect.

- **The fault is an emergent property of a particular combination of aspects woven into**

the base program. This is a more insidious version of the third alternative, but compounded by the integration and interaction of data and control dependences from multiple aspects combined with those occurring in the base program. The fault may or may not exist with a different combination of aspects with respect to the base program.

Störzer [46] discussed several problems that can show up in AspectJ programs. Interference on dynamic binding can happen when a method is introduced into a class in a hierarchy. The result of a dynamic lookup may be the newly introduced method instead of the method originally dispatched. Use of the *declare parents* construct to move a class down the inheritance hierarchy can also result in binding interference. Another problem with changing the inheritance hierarchy is that the type of a class will change. Additional up-casts are suddenly allowed, and use of the Java *instanceof* operator might return *true* where it previously returned *false*. *Pointcut specifications* are problematic since the use of wildcards can easily miss necessary or accidentally include unwanted join points. *Advice* can possibly modify both state and control flow of a program. The modifications do not even need to be direct — advice can influence some distant object through a sequence of method calls. When using several aspects, they might interfere with each other directly by introductions, precedence and advice, but also indirectly by manipulating and reading the same state of the base program.

Störzer and Krinke [48] later also considered interface introduction, where default implementations for interface methods are provided. This can result in "forgotten" implementations, and flaws in the program. That is, a class implements an interface, but "forgets" to redefine all default implementations.

Alexander et al. [9] proposed a candidate fault model for AspectJ programs that describes the following fault types:

- **Incorrect strength in pointcut patterns.** The strength of the pattern in the signature of a pointcut determines which join points are selected. If the pattern is too strong, some necessary join points will not be selected. If the pattern is too weak, additional join points will be selected that should be ignored.

- **Incorrect aspect precedence.** The order in which advice from multiple aspects are executed, affects the system behavior, especially when there are mutual interactions between aspects through state variables in the base program.

- **Failure to establish expected postconditions.** Clients expect method postconditions to be satisfied regardless of whether or not aspects are woven into the base code. Thus, for correct behavior, woven advice must allow methods in the base program to satisfy their postconditions.

- **Failure to preserve state invariants.** In addition to establishing postconditions, methods must also ensure that state invariants are satisfied. Woven advice might cause violations of state invariants.

- **Incorrect focus of control flow.** Sometimes join points should only be selected in a particular execution context, e.g. only the top level of a method call should be selected, not consecutive recursive calls. Failure to restrict execution to the proper context could result in a failure that is difficult to diagnose.

- **Incorrect changes in control dependencies.** Around advice can significantly alter the behavioral semantics of a method. Thus, defects may arise from assumptions on control dependencies (and data dependencies) that are not longer valid in the woven code.

Ceccato et al. [17] extended this fault model in with three new fault types:

- **Incorrect changes in exceptional control flow.** An advice that throws an exception might cause an implicit modification of the control flow, because the exception triggers the execution of a catch statement, either in the aspect itself or in the base program. Moreover, if exception softening is used, different branches might be taken in the original and the composed code.

- **Failures due to inter-type declarations.** Inter-type declarations could produce ripple effects in the control flow, each time the control flow depends on the static class structure.

- **Incorrect changes in polymorphic calls.** Modification in the system behavior may occur when a method introduction is used to override a method inherited from a super class. Before weaving the aspect, any invocation to such a method was redirected to the method in the super class, while after weaving, the same invocation, is dispatched to the introduced method.

Koppen and Störzer [29], and later Störzer and Graf [47], discussed what they refer to as *the fragile pointcut problem* for most current AOP pointcut languages. When considering system evolution, using pointcuts based on wildcards and naming conventions can easily lead to spurious or missed matches of join points, especially when the base code evolve. The semantics of the pointcuts are *silently altered*. The problem can result from renaming, moving, deleting or adding classes, methods and fields in the base code. The authors proposed a static analysis technique for detecting these changes. While the authors' work is important, they discussed pointcuts in a context of evolution only. The technique can be used to compare two versions of a system, but cannot help with verifying that the correct join points where matched in the first place.

van Deursen et al. [50] also proposed an aspect-oriented fault model, targeting AspectJ-like languages.

- **Faults due to inter-type declarations:**
  - *Wrong method name in introduction*, leading to missing or unanticipated method override.
  - *Wrong class name in member introduction*, leading to a method body in the wrong place in the class hierarchy.

    - *Inconsistent parent declaration*, resulting in a subclass that violates Liskov's and Wing's behavioral notion of subtyping [33] and/or Meyer's design by contract rules for inheritance [37].
    - *Inconsistent overridden method introduction*, also resulting in violation of behavioral subtyping.
    - *Omitted parent interface*, resulting in a method that was intended to implement an interface method, but which now stands on its own.

- **Faults in pointcuts:**
    - *Wrong primitive pointcut*, e.g. using *call* instead of *execution*.
    - *Errors in the conditional logic combining the individual pointcut conditions*.
    - *Wrong pointcut pattern*, especially with the use of wildcards, and when the underlying classes are modified.

- **Faults in advice:**
    - *Wrong advice specification* (e.g. using before instead of after).
    - *Wrong or missing proceed in around advice*.
    - *Wrong or missing advice precedence*.
    - *Advice code causing a method to break its class invariant or fail to meet its postcondition*.

McEachen and Alexander [36] investigated problems resulting from the unanticipated composition of aspects and base classes that can arise when foreign aspects are rewoven with AspectJ. A *foreign aspect* is an aspect written by a third party and for which we do not have access to the source code. The problem is due to AspectJ's option for creating class files containing annotations that enable later *reweaving*, combined with the use of *unbounded pointcuts*, pointcuts that are not defined such that they only match join points from packages and classes present in the original environment that the aspect was specifically developed and tested for. McEachen's and Alexander's work can be seen as a special case fault model for the evolution and reweaving of AspectJ programs.

2.2.2 Testing of Aspect-Oriented Programs

Xu and Xu [57] proposed an approach to test generation for aspect-oriented programs based on aspect-oriented UML models. Such a model extends the basic UML and consists of class diagrams, aspect diagrams and sequence diagrams. The approach aims to adequately exercise interaction between aspects and classes. "Woven" sequence diagrams are created that include both methods and woven advice. From a woven sequence diagram and desired coverage criteria (polymorphic and branch coverage), a flow graph is generated that provides a testable model of class and aspect behavior. This flow graph is then transformed into a flow tree, where each path from a leaf node to the root indicates a test case. Xu and Xu's approach is implicitly based on the observation that aspects can add extra message sequences and change the ordering of message sequences in the base program, and the assumption that faults could occur by aspects doing this.

Xu et al. [54] proposed a state-based approach to testing aspect-oriented programs using the FREE state model developed for testing object-oriented programs [15]. The FREE model is extended to an *aspectual state model* to deal with aspect-oriented constructs. Once a state model is created, it can be transformed into a transition tree, in which each path from the root to a terminal node (i.e., a sequence of transitions) is a test requirement. A test requirement becomes a test case when the variables are assigned specific values for the corresponding conditions. A test suite using this approach can achieve N+ coverage, which will reveal all state control faults, all sneak paths and many corrupt state bugs. It can also reveal some faults specific to aspects, including incorrect strength in pointcut patterns and failure to preserve state invariants, as described in [9].

In [53], Xu and Xu extended the state-based approach to *incremental* testing of aspect-oriented programs. Aspects are seen as incremental changes to a base program, which can introduce new object states and transitions, and remove and update existing transitions. As such, aspects may lead to subtle differences in the sequence of messages that can be accepted by the base class objects, and aspect-oriented faults will likely result in unexpected object states and transitions. The incremental testing process involves first testing the base program with a state-based approach, and then create test suites for the woven program by reusing, modifying and extending base program test cases. Xu and Xu argued that their technique will help detect at least four aspect-specific fault types: pointcuts picking out extra join points, pointcuts missing join points, incorrect advice types (e.g. *before* instead of *after*), and incorrect advice implementation.

Recently, Xu and Xu [58] discussed their state-based technique for testing so-called *integration aspects*, which are aspects that compose classes that implement separate concerns. The state-based approach is essentially the same as for incremental testing of aspects, but used for of another kind of aspect-oriented programs (i.e., aspects as integrators of classes rather than as increments on classes). In a case study their approach detected pointcuts picking out extra join points, pointcuts missing join points, incorrect advice types and traditional faults in advice bodies.

In addition to the fault model described in Section 2.2.1, Ceccato et al. [17] proposed two coverage criteria to help expose AOP specific faults. The *designator coverage* criterion is used to expose faults due to incorrect focus of control flow, e.g. resulting from the use of the *cflow* pointcut of AspectJ. The *cflow* pointcut cannot be evaluated statically but requires evaluation of the run-time execution stack. The designator coverage criterion requires that every feasible execution stack associated with the pointcut is exercised by some test case. The *composition coverage* criterion is used to detect incorrect aspect precedence, by requiring every possible precedence configuration to be tested that changes at least one data dependence with respect to any of the previously tested configurations. The authors proposed an adaption of the branch coverage criterion to reveal incorrect strength in pointcut patterns, and an adaption of data-flow criteria to deal with failures to establish postconditions and preserve state invariants. An adapted version of the branch coverage criterion was also proposed to expose faults coming from inter-type declarations and changed control flow.

Mortensen and Alexander [40] proposed combining coverage and mutation testing to adequately test AspectJ programs. An aspect code fragment is covered by *statement coverage* if every statement through the fragment is executed at least once after being woven into the program. With *insertion coverage*, each aspect code fragment is tested at each point it is woven into the program. *Context coverage* extends insertion coverage to test an aspect code fragment in each place it is *used*. *Def-use coverage* tests def/use pairs within advice, between different advice fragments, between advice and methods and between methods where control flow has changed due to advice control flow changes. The coverage criteria can be used for exposing failures to establish postconditions and preserving state invariants, incorrect focus of control flow, and incorrect changes in control dependences. Two mutation operators, *pointcut strengthening* and *pointcut weakening*, are used to detect incorrect strength in pointcut patterns. The *precedence changing* mutation operator is used to detect incorrect aspect precedence.

Lemos et al. [32] proposed a technique for structural unit testing of AspectJ programs using *aspect-oriented def-use data flow graphs* (AODUs) and woven bytecode. The authors described several control flow and data flow testing criteria based on the coverage of nodes and edges in the AODU: the *all-nodes criterion*, the *all-crosscutting-nodes* criterion, the *all-edges criterion*, the *all-crosscutting-edges criterion*, the *all-uses criterion* and the *all-crosscutting-uses criterion*. Crosscutting nodes, edges and uses are related to the execution of advice. Lemos et al. argued that e.g. their all-nodes criterion could discover the fault types of selecting unintended join points, missing intended join points, and incorrect advice execution order, faults types described in [9].

van Deursen et al. [50] also described testing criteria for their fault model (described in Section 2.2.1). For introduction of methods, statement or branch coverage should apply, in addition to exercising all possible polymorphic bindings. For changes to the inheritance hierarchy, adequacy criteria for polymorphic calls should be used. For pointcuts with signatures and patterns involving wildcards, a form of traditional boundary testing should be used, and for pointcuts composed with conditional logic, every relevant condition combination should be exercised. Test adequacy for advice can be based on statement or branch coverage. Advice should be exercised at each join point where the advice is activated, but fully exercising all branches is only needed at one join point.

Anbalagan and Xie [11] recently proposed a framework for automated testing of pointcuts in AspectJ programs. The framework receives as input a threshold value and a list of source files, including the source of aspects and target classes. The framework outputs a list of matched join point in the target classes as well as a list of boundary join points, which are join points that do not satisfy the pointcut expression but are "close" to the matched join points. These boundary join points are identified as those unmatched join point candidates whose distance from the matched join points (measured in terms of the number of edit operations necessary to transform one into the other) are less than a predefined threshold value. A developer could inspect the matched join points and boundary join points to determine the correctness of the pointcuts.

Lemos and Lopes [31] also recently proposed an approach for pointcut testing, and provided a classification of pointcut faults. A pointcut can be wrong in one of the following ways: 1) it selects some of the intended join points but also some unintended, 2) it selects none of the intended join points, 3) it selects all the intended join points but also unintended ones, and 4) it selects some of the intended join points but not all of them. In order to detect unintended join points, all join points currently selected by a pointcut are gathered, and the methods in which they are located are integrated with the pointcut's associated advice. If the integration fails, it may indicate that the join point and the advice do not "belong together" and the pointcut might be faulty. To detect neglected join points, the authors propose using mutation operators to simulate faults that result in restricting the set of matched join points.

Several methods and associated tools for automatic generation of tests for aspect-oriented programs have been published that are not based on fault models or explicit fault-directed testing.

Examples are Zhao’s data-flow based testing technique [60], the JAOUT framework by Xu et al. [56], and a framework based on wrapper classes proposed by Xie and Zhao [52]. Recently, Xu [55] and Zhao et al. [61] proposed approaches for regression testing of aspect-oriented programs.

There has been done very little evaluation of proposed testing approaches, but Naqvi et al. [42] did an informal comparison of three AOP testing techniques in light of the fault model of Alexander et al. [9]. Zhao’s data-flow approach [60], the state-based approach by Xu et al. [54], and a third approach combining state-based testing and flow graph based testing [59] were considered. None of the techniques were considered good at revealing the fault types of the fault model, but the authors believed the state-based approach to have the greatest potential.

CHAPTER THREE

EFFECTS OF POINTCUTS AND ADVICE ON PROGRAM DEPENDENCES

3.1 Introduction

When discussing faults, it is interesting to know what possible effects on program execution those faults can have. The behavior of a program is bound by the *control dependences* and *data dependences* present in the program, and pointcuts and advice can, and often do, affect control- and data dependences of programs. As we will see in the next two chapters, many types of faults can result in changes in these dependences in a program, which in turn can affect control- and data flow through the program.

In existing implementations of AspectJ [5, 1], pointcuts technically do not exist at run-time. Instead, they are considered weaving rules that transform plain Java code and aspect code into Java byte code at compile- or load time. For pointcuts that can be determined statically, the associated advice code is inserted at the appropriate places in byte code; for pointcuts that need run-time evaluation, conditional checks are also inserted. Advice is usually transformed into Java methods in byte code, but we can still think of them as advice. This will keep the discussion at the level of abstraction of AspectJ language semantics rather than Java byte code.

This chapter provides only examples of how advice can affect program dependences; we do not claim to cover every situation possible.

3.2 Effects on Control Dependences

In this section a pointcut is considered to have caused a piece of advice to be woven at some join point, and possible effects on control flow and control dependences are shown through *control flow graphs* (CFGs) of example weavings.

A control flow graph is a directed graph that consists of a set N of nodes and a set $E \subseteq N \times N$

of directed edges between nodes [62]. Each node represents a linear sequence of statements (a *basic block*). Each edge representing transfer of control is an ordered pair (n_1, n_2) of nodes, and is associated with a predicate that represents the condition of control transfer from node n_1 to node n_2. In a flow graph there is a *begin node* and an *end node* where the computation starts and finishes, respectively. The begin node has no inward edges and the end node has no outward edges. Every node in a flow graph must be on a path from the begin node to the end node. To model exceptional control flow, a second end node may be added, representing the end of execution as the result of the throwing of an exception. In this case, every node must be on a path from the begin node to one of the end nodes.

Let $start$ be the begin node, and $exit$ be an end node of a CFG. A node y is *control dependent* on x if from x we can branch to node u or node v; from u there is a path to to $exit$ that does not include y, and from v every path to $exit$ includes y [12].

The following CFGs are somewhat simplified compared to the above definition. Predicates are not explicitly associated with edges. Instead, the result of the evaluation of a conditional node is associated with some edges. For example, if the condition of a node is *if (i > 5) proceed*, then one outgoing edge is marked with "proceed" while the other is not. The condition itself is not shown as only the possible outcomes matter. Edges representing returns (i.e., end of execution) from advice and join points are marked with "return".

Consider the control flow fragment depicted in the CFG of Figure 3.1. The nodes $start$ and $exit$ represent the start and exit points of the fragment, correspondingly. There is also a join point j. Solid edges represent flow of control and the dotted edge represents a control dependence. As $start$ is a conditional node, the execution of j is control dependent on $start$. The $exit$ node, however, will execute in any circumstance and is not control dependent on neither $start$ nor j. The edge from $start$ to $exit$ is there simply to have some control dependence for sake of example.

In Figure 3.2(a) a piece of *before* advice a is woven into the program fragment. Advice are represented as boxes in this and the following CFGs. A control dependence is added from a

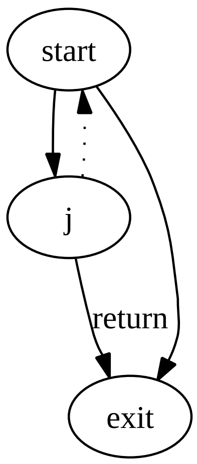
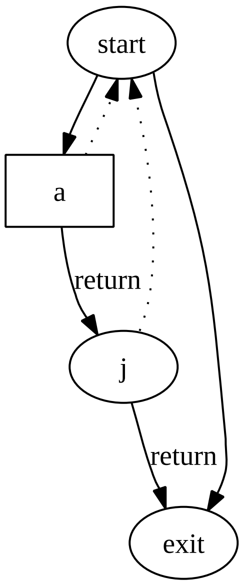


Figure 3.1: CFG of unwoven program

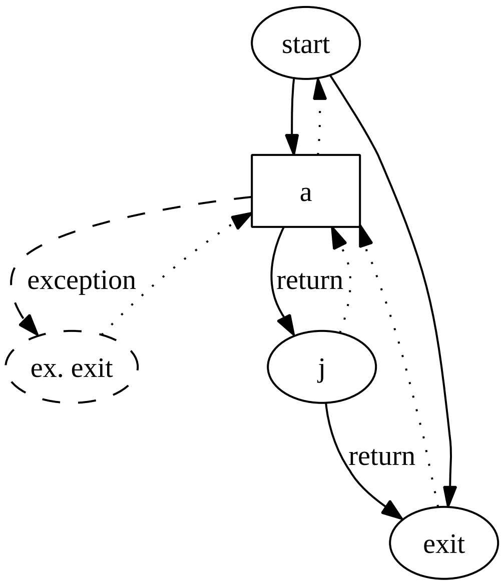
to $start$, but otherwise there is no change in dependences. Figure 3.2(b) depicts advice a that conditionally throws an exception. “Conditionally throws” means that the advice may or may not throw an exception. Exceptional control flow is shown with dashed edges. Throwing an exception not only will bypass the j, but the rest of the control flow fragment as well, and the fragment will return through the node *ex. exit* (exceptional exit) instead of $exit$. The possibility of an exceptional return from a implies a change in control dependences. Both j and $exit$ become dependent on a after weaving. In Figure 3.2(c) a *unconditionally* throws an exception, having the effect of j never being executed, and $exit$ being control dependent on $start$. The node j and incident edges are colored in gray to show that they are never reached.

Figure 3.3 shows the similar situations for *after* advice. In Figure 3.3(a) no exception is thrown and no control dependences are altered. In Figure 3.3(b) a conditionally throws an exception, making $exit$ dependent on a. Note that in this case, j is still dependent on $start$. In Figure 3.3(c), a unconditionally throws an exception, meaning $exit$ becomes control dependent on $start$, and j is still control dependent on $start$.

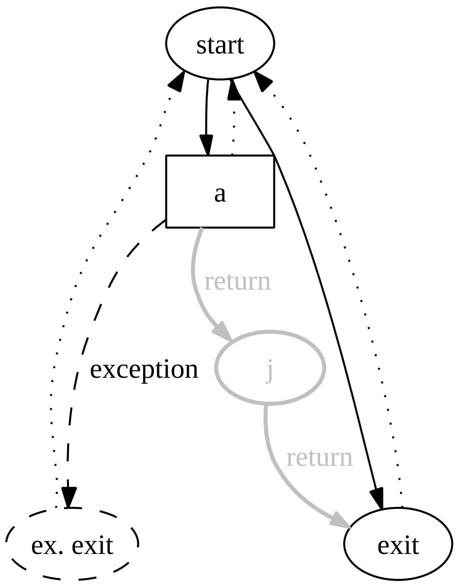
For *around* advice there are a greater number of situations that can occur, since around advice *may* or *may not* call *proceed* in addition to throwing exceptions. In Figures 3.4–3.7, $a1$ and $a2$ are both parts of the same around advice a. If the advice contains a *proceed* statement, $a1$ corresponds



(a) No exception thrown

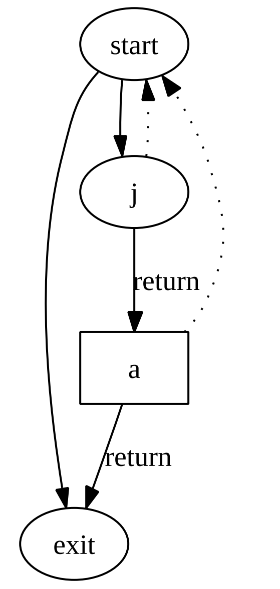


(b) Exception conditionally thrown

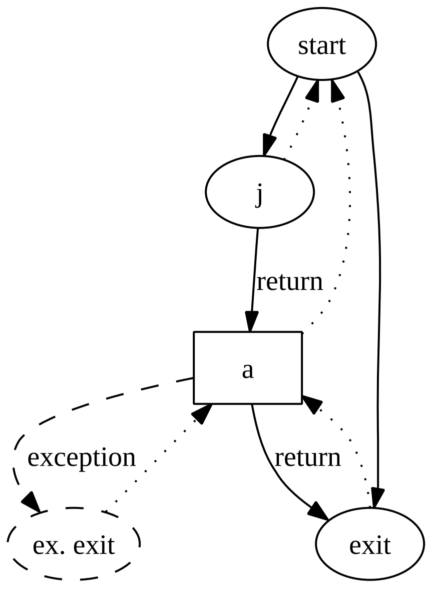


(c) Exception unconditionally thrown

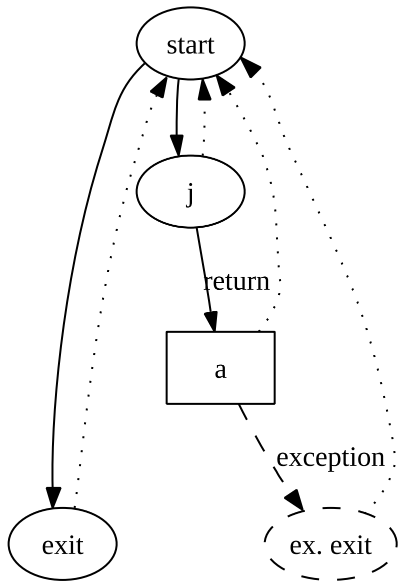
Figure 3.2: Effects of throwing exceptions in before advice



(a) No exception thrown



(b) Exception conditionally thrown



(c) Exception unconditionally thrown

Figure 3.3: Effects of throwing exceptions in after advice

to the code before *proceed*, and $a2$ corresponds to the code after *proceed*. If there is no *proceed* statement, $a1$ and $a2$ can be chosen arbitrarily, as long as the concatenation $a1a2 = a$ holds. In the CFGs presented there is at most one *proceed* statement. In reality there can be any number of *proceed* statements in a piece of advice, but keeping the number to at most one keeps the discussion as clear as possible and does not affect its generality.

In Figure 3.4(a) $a1$ unconditionally calls *proceed*, and no control dependences are affected. In Figure 3.4(b) *proceed* a contains no *proceed* statement, effectively canceling j. No control dependences are otherwise altered. In Figure 3.4(c) $a1$ conditionally calls *proceed*, which makes j control dependent on $a1$.

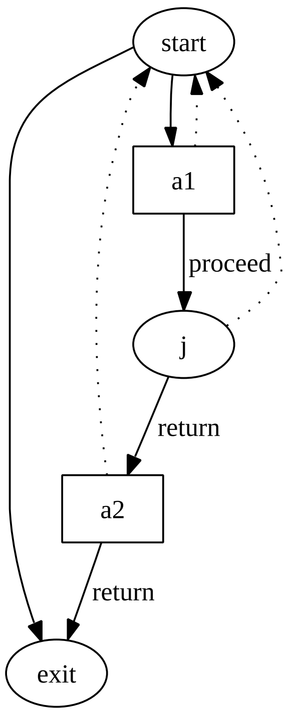
Around advice may throw exceptions just as before and after advice may. In Figure 3.5(a) and 3.5(b), $a1$ and $a2$ conditionally throws exceptions, correspondingly. In the former case both j, $a2$ and $exit$ are made control dependent on $a1$, and in the latter case $exit$ is made control dependent on $a2$. In both cases, *proceed* is unconditionally called.

In Figure 3.6(a) $a1$ unconditionally throws an exception, meaning that any call to *proceed* and hence j and $a2$ will never execute. Furthermore, $exit$ is made control dependent on $start$. When $a2$ unconditionally throws an exception, as in Figure 3.6(b), $exit$ is also made control dependent on $start$, but otherwise no dependences are altered.

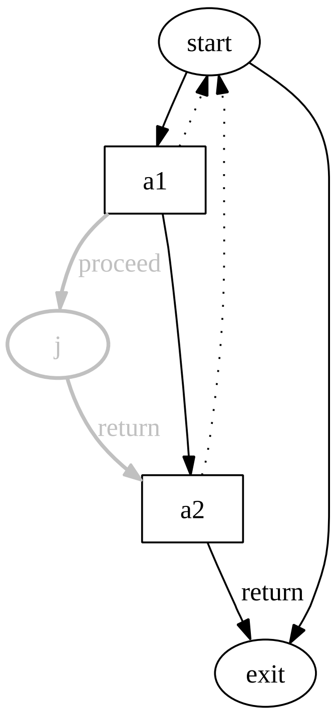
Finally, Figure 3.7 depicts the combinations of both conditionally calling *proceed* and conditionally throwing an exception. In Figure 3.7(a) $a1$ throws an exception, making j, $a2$ and $exit$ control dependent on $a1$. In Figure 3.7(b) $exit$ is control dependent on $a2$ for it possibly throwing an exception, whereas j is made control dependent on $a1$ because of the conditional call to *proceed*.

3.3 Effects on Data Dependences

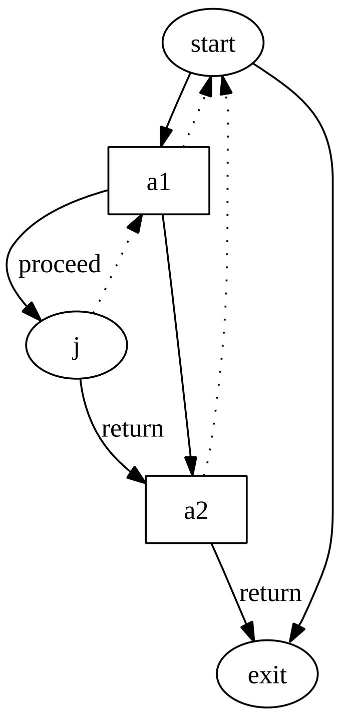
Just as advice may affect the control flow and control dependences of a program, advice may also change data flow and data dependences.



(a) Unconditional call to *proceed*

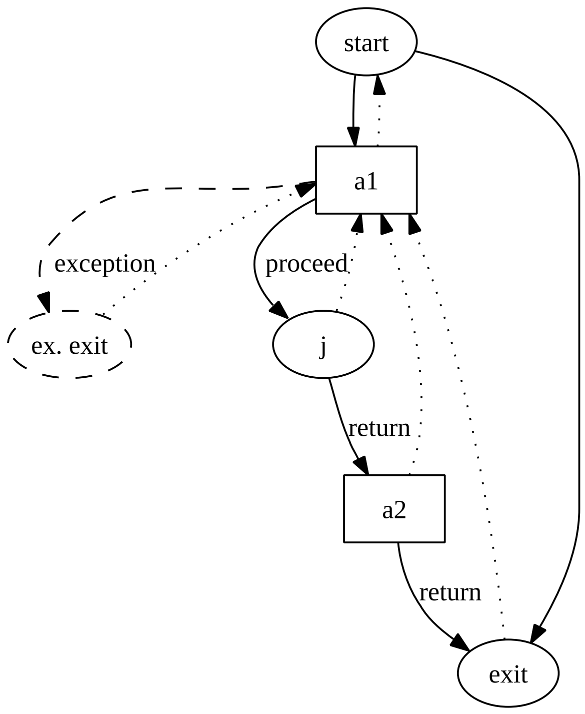


(b) No call to *proceed*

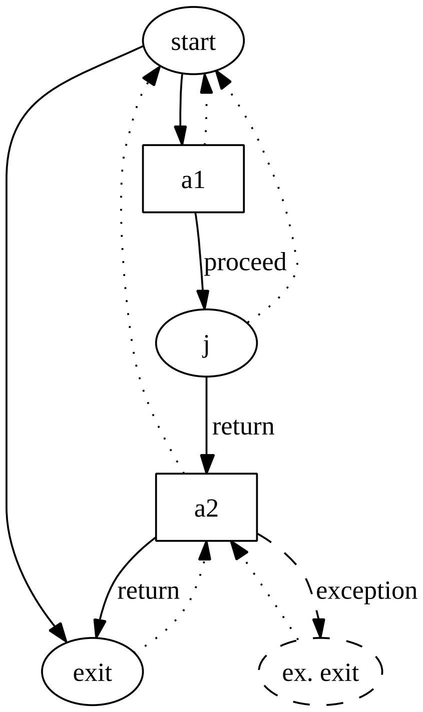


(c) Conditional call to *proceed*

Figure 3.4: Effects of calling and not calling *proceed* in around advice

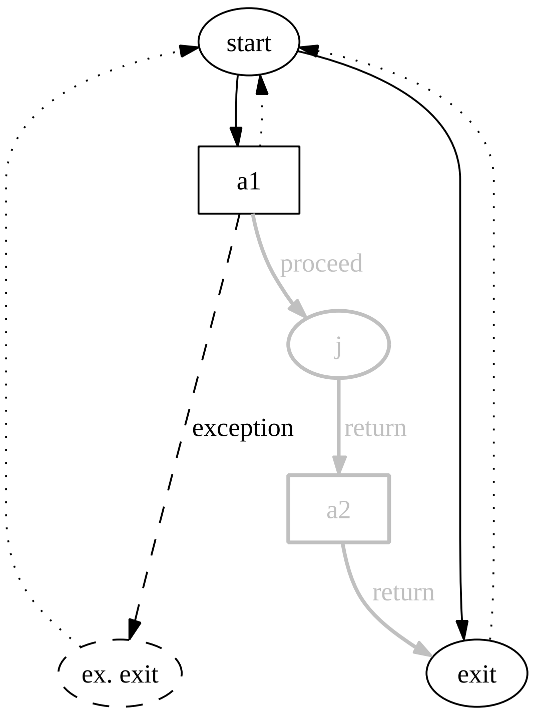


(a) Exception thrown in upper around block

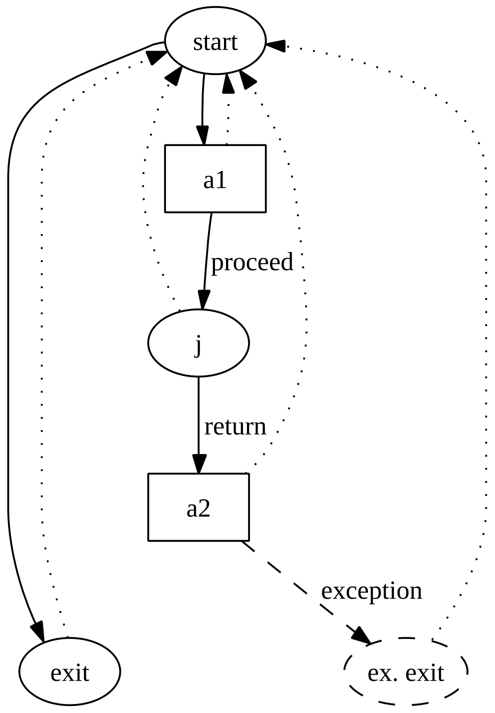


(b) Exception thrown in lower around block

Figure 3.5: Effects of conditionally throwing exception in around advice, with *proceed* unconditionally called

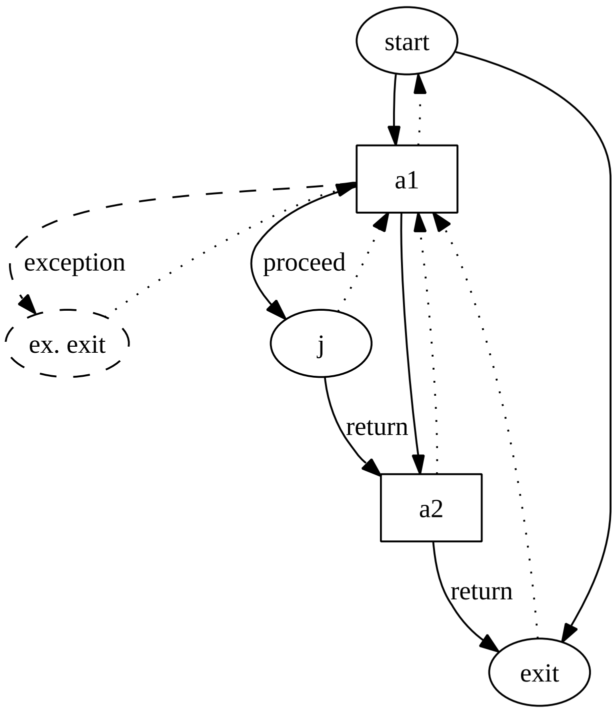


(a) Exception thrown in upper advice block

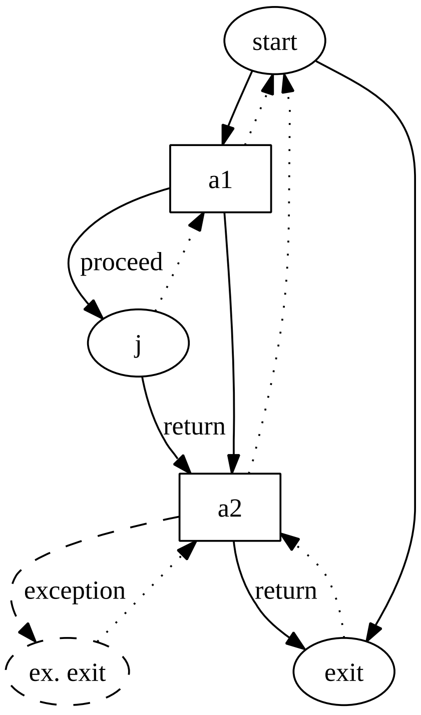


(b) Exception thrown in lower advice block

Figure 3.6: Effects of unconditionally throwing exception in around advice, with *proceed* unconditionally called



(a) Exception thrown in upper advice block



(b) Exception thrown in lower advice block

Figure 3.7: Effects of conditionally throwing exception combined with conditionally calling *proceed* in around advice

An assignment to a variable *defines* that variable. The occurrence of a variable on the right-hand side of an assignment (or in other expressions) *uses* the variable. We can speak of the *def* of a variable as the set of graph nodes that define it; or the *def* of a graph node as the set of variables that it defines; and similarly for the *use* of a variable or graph node [12]. We denote the def of a node as *def(n)*, where n is the node.

In the following figures, we augment the CFGs with *def(n)* and *use(n)* in the nodes that define or use variables. Specifically, we consider two nodes s and s' in the CFG of some program fragment. From s there is a path that eventually ends up at the join point j. From j there is a path that eventually ends up at s'. To illustrate how advice can affect data dependences, definitions and uses of a variable x in s, s' and advice a are considered in various combinations.

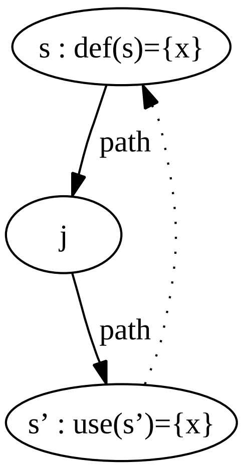
Consider Figure 3.8(a), which depicts an unwoven program P_1. The node s defines x, while s' uses x. There is a data dependence from s' to s (denoted by a dotted edge), meaning that there is no definition of x on the path from s to s'. We say that the path is *definition-clear* [62], or that the definition in s *reaches* the use in s'.

In Figure 3.8(b), before advice a is woven into the program, and a has a definition of x. The result is that s' is no longer data dependent on s since the path from s to s' is no longer definition-clear. Instead, there is now a data dependence from s' to the advice a.

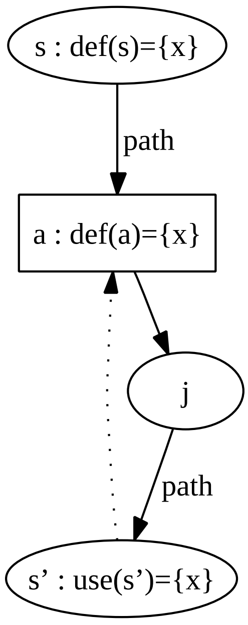
Figure 3.8(c) illustrates the similar situation for after advice a, which has a definition of x. The node s' is made dependent on a instead of s.

In Figure 3.9 around advice $a = a1a2$ has a definition of x. In Figure 3.9(a) the definition is in $a1$, and s' is made data dependent on $a1$ rather than s. In Figure 3.9(b) the definition is in $a2$ and s' is made data dependent on $a2$ instead of s.

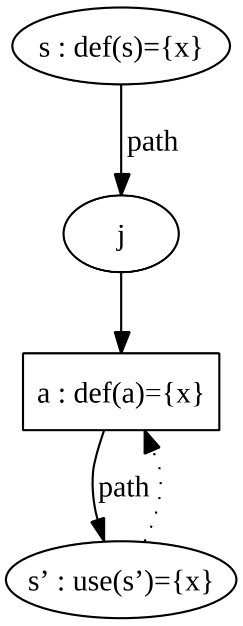
Now consider the unwoven program fragment P_2 depicted in Figure 3.10(a). It is similar to P_1, but the use of x is at the join point j rather than in node s'. In Figure 3.10(b) before advice a is woven into the program, and a has a definition of x. The data dependence between j and s no longer exists. Instead j is now data dependent on the advice a. Figure 3.10(c) shows the similar



(a) Unwoven program P_1

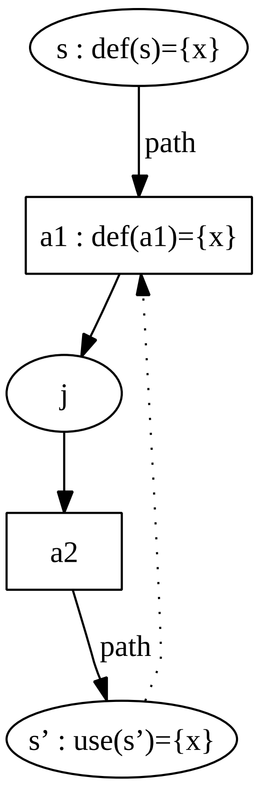


(b) Before advice defines x

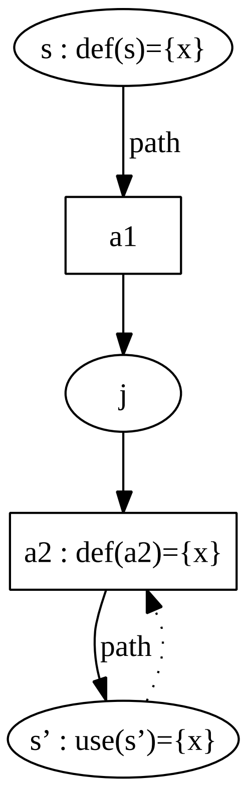


(c) After advice defines x

Figure 3.8: Effects on P_1 of data definitions in before and after advice



(a) Upper block of around advice defines x

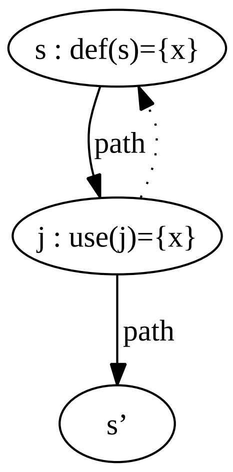


(b) Lower block of around advice defines x

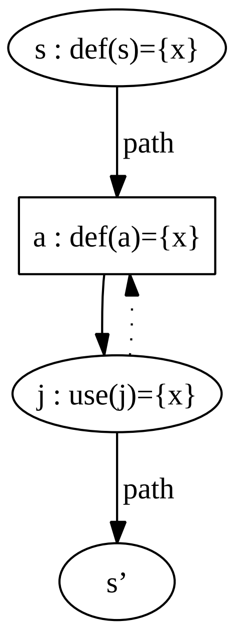
Figure 3.9: Effects on P_1 of data definitions in around advice

situation where around advice a is woven into the program and the upper block $a1$ of the advice defines x. The join point j is made dependent on $a1$ instead of s.

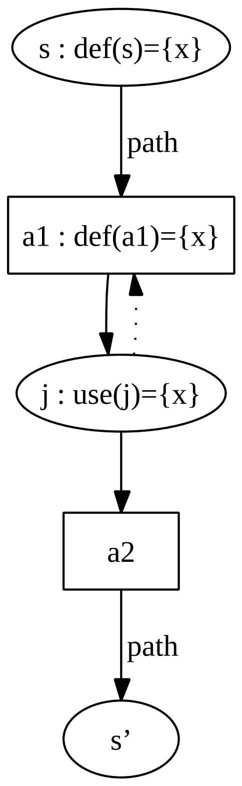
The unwoven program fragment P_3 in Figure 3.11(a) is similar to P_2, but now the definition of x is in j and the use is in s'. The addition of after advice and around advice defining x results in changes in data dependences analogous to the changes in P_1 and P_2.



(a) Unwoven program P_2

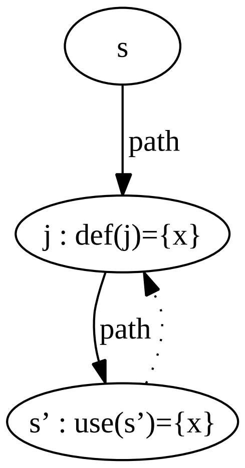


(b) Before advice defines x

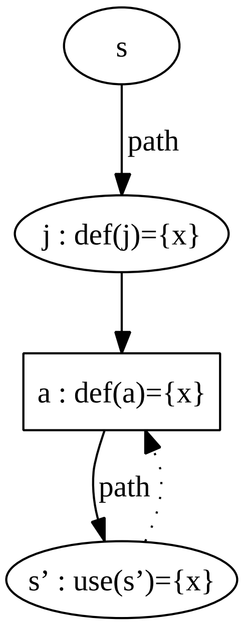


(c) Around advice defines x

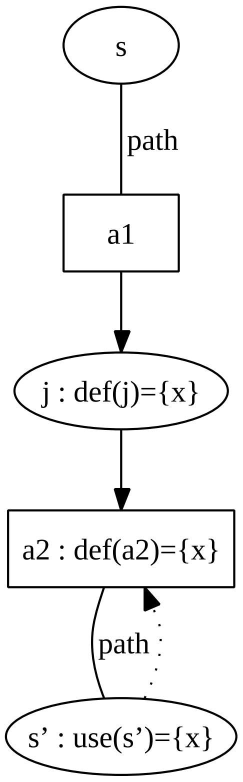
Figure 3.10: Effects on P_2 of data definitions in before and around advice



(a) Unwoven program P_3



(b) After advice defines x



(c) Around advice defines x

Figure 3.11: Effects on P_3 of data definitions in after and around advice

CHAPTER FOUR

POINTCUT FAULTS

4.1 Introduction

This chapter presents an interpretation of the fault/failure model [51], for pointcut faults, also building on the RELAY model [44] in its treatment of “potential errors” and “context errors.” A classification of identified pointcut fault types is also presented, and the individual fault types described.

4.2 Fault/Failure Model for Pointcuts

As described in Chapter 2, the fault/failure model [51] comprises three individually necessary and collectively sufficient conditions for a fault to produce a failure:

- The fault must be executed (*execution*).
- The succeeding data state must be infected (*infection*).
- The data-state error must propagate to the output (*propagation*).

This section presents an interpretation of the fault/failure model for pointcuts faults. A necessary and sufficient condition for a pointcut fault to execute is presented, and three different kinds of errors that can result from a pointcut fault are identified. A necessary and sufficient condition for a pointcut fault to cause an infection is also presented, as well as several necessary conditions for an infection to propagate to the output and cause a failure.

4.2.1 Assumptions

Most pointcuts cannot affect any concrete state of the program, such as the value of a variable. In other words, they are *side-effect free*. The only exception is the *if* pointcut. The argument to an *if* pointcut is a regular Java boolean expression, and can as such affect program state as a side-effect, for example by calling a method as part of that boolean expression. *If* pointcuts are, however, assumed side-effect free in the following discussion. Since evaluation order among pointcuts is undefined [27], considering side-effects in the fault/failure model would make it overly complex without providing much benefit since the *if* is a very small part of the AspectJ language and faults related to side effects account for a very small part of the fault model. Leaving this special case out clarifies the model and still accurately models the other primitive pointcuts of the language. Using *if* pointcuts with side-effects is in any case considered bad programming practice.

In a discussion of pointcuts used for the purpose of advice only, we can assume that every pointcut is associated with one or more pieces of advice. Execution order among advice is defined, and is under full control of the programmer(s).

4.2.2 Execution

To make use of the fault/failure model for pointcuts, some notion of execution of a pointcut is needed. It is natural to think of the execution of a pointcut, and hence a pointcut fault, as the evaluation of that pointcut. However, the evaluation order of individual pointcuts as well as the parts of a complex pointcut is undefined [27]. Some pointcuts might be evaluated at statically at weave time, other pointcuts must be evaluated at run-time. We cannot make any assumptions about which are evaluated statically and which are evaluated dynamically; neither can we make any assumptions about the relative order among static evaluations or the relative order among dynamic evaluations, since the language leaves these orders undefined [27]. Thus, the strength of the statements we can make about when and if a pointcut fault is executed is limited. In the following, a conservative approach will be used where a pointcut is considered to be evaluated as a join point occur, since at that point, it must have been fully evaluated[1] (if not, it could not possibly decide to select that join point or not).

For a fault in a pointcut to possibly have an effect, it must first be evaluated and given the

[1]However, no assumptions are made on the relative evaluation order among pointcuts.

chance to select/not select and possibly expose context. This can be expressed as a necessary and sufficient condition for a pointcut fault to be executed.

Pointcut Fault Execution Condition. *A pointcut fault is executed if and only if the simplest pointcut expression containing that fault is evaluated.*

A *simple pointcut expression* is a pointcut expression that is not built up from other pointcut expression. Consider for example:

```
pointcut p1(Foo f) : call(public * bar(..)) && this(f);
```

The expressions `call(public * bar(..))` and `this(f)` are both simple pointcut expressions, while `call(public * bar(..)) && this(f)` is a complex pointcut expression built from the two simple ones. If, however, there is a fault in the complex expression, e.g. `&&` should have been ||, the *simplest pointcut expression containing that fault* is the expression `call(public * bar(..)) && this(f)`.

It is important that the simplest pointcut expression containing the fault is evaluated, since evaluating a complex pointcut does not necessarily result in evaluating each of the simpler pointcuts the complex pointcut is composed of. This is analogous to for example short-circuit evaluation of boolean expressions in other languages such as C or C++.

4.2.3 Infection

The second condition of the fault/failure model states that after execution of a fault, a succeeding data state must be infected. For pointcuts, *data state* must be interpreted in an abstract way, since evaluating a pointcut does not affect any variables or other programmer-observable state in the program (except for context exposure which assigns values to formal parameters of pointcut and advice) Still, the outcome of the evaluation must be stored somewhere and can be considered part of the program state.

Some pointcuts, such as *call* and *execution*, only select join points, while other pointcuts both select join points and expose context at those join points, e.g. *this* and *target*. For selection, there are two possible outcomes of evaluating a pointcut expression at a join point; either the join point is selected, or it is not. Either outcome can be correct or incorrect. We have the following cases:

- The join point was selected, as it was supposed to.
- The join point was not selected, and neither was it supposed to.
- The join point was selected, although it was not supposed to.
- The join point was not selected, although it was supposed to.

In summary, there are two possible types of selection errors that can be the result of a pointcut evaluation: *positive selection error* and *negative selection error*.

Definition 1. *A* ***positive selection error*** *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.*

Definition 2. *A* ***negative selection error*** *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.*

Note that at these errors are the result of evaluating the *simplest* pointcut expression containing a fault. An error at this level is what the RELAY model calls a *potential error*. At this level, the decision whether advice will execute at the join point or not, might not yet have been decided. For that to happen, the error must transfer to a pointcut expression that is actually on the right-hand side of some advice.

Context exposure involves a pointcut expression on the right-hand side of a pointcut or advice binding one of the formal parameters on the left-hand side of the pointcut (or the advice) to context at the join point. After the evaluation of a pointcut expression that exposes context to a formal parameter, there are two possible results (not considering selection):

- The parameter contains the correct value (i.e., a reference to the right object, or an intended $null$ value)

- The parameter contains an incorrect value (i.e., a reference to the wrong object, or an unintended $null$ value)

A parameter might be assigned a $null$ value, depending on the primitive pointcut and the available context at the join point (see Section 2.1.1.2), and if the pointcut does not select the join point, values might not be assigned to the parameters.

Hence, there is one possible type of exposure error that can result from the evaluation of a pointcut expression.

Definition 3. *A **context exposure error** 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.*

Consider the following example incorrect pointcut:

```
pointcut p1(Foo f) : call(public * bar(..) && this(f);
```

which should have been

```
pointcut p1(Foo f) : call(public * bar(..) && target(f);
```

That is, the incorrect pointcut exposes the *current object* (using *this*) instead of the *target object* of the call (using *target*), which may result in the parameter f being assigned the incorrect value, i.e., a reference to the wrong object.

A necessary and sufficient condition for a pointcut fault to cause an infection can now be formulated.

Pointcut Fault Infection Condition. *A fault in a pointcut expression of a named pointcut or a piece of advice causes an infection at some join point if and only if the fault is executed, and the* *execution of the fault results in either 1) the join point being selected and it not being intended to be selected, 2) the join point not being selected and it being intended to be selected, or 3) a formal parameter on the left-hand side of the pointcut (or advice) containing an incorrect value.*

Consider the following example program:

```
public class C {
   public static void main(String[] args) {
      C c = new C();
      c.foo();
      c.foo(new Bar());
   }
   public void foo() {
      \\ some computation
   }
   public void foo(Bar b) {
      \\ some computation
   }
}
public aspect A {
   pointcut p1() : call (public void foo());
      \\ incorrect; should have been call (public void foo(Bar))
   before() : p1() {
      \\ some computation
   }

   pointcut p2() : call (public void foo(Bar));
      \\ incorrect; should have been call (public void foo())
   before() : p2() {
      \\ some computation
   }

   pointcut p3(Bar b): call (public void foo(Bar)) && target(b);
      \\ incorrect; should have been call (public void foo(Bar)) && args(b)
   before(Bar b) : p3(b) {
      \\ some computation
   }
}
```

Here, $p1$, $p2$ and $p3$ are all incorrect. When the call `c.foo()` is performed in the main method, $p1$ will select the join point although it was not intended to. The pointcut $p2$ will not select this call, since it specifies that the parameter of the call must be of type Foo. This is also unintended.

Finally, in $p3$, if the target object is of type Bar, the call `c.foo(new Bar())` will be selected, as intended, even though the pointcut is incorrect. That is, there is no selection error. However, the *target* and the *args* pointcut most likely will expose different objects, resulting in the formal parameter b of $p3$ having an incorrect value. All these situations are examples of infection.

4.2.4 Propagation

An infection means that after evaluating a pointcut expression containing a fault, there is a data state error. However, a pointcut cannot itself produce any observable output[2], so there cannot yet be a failure. For a pointcut error to propagate to observable output and produce a failure, several conditions must be met.

4.2.4.1 Propagation from Pointcut to Advice

Pointcut Fault Propagation Condition 1. *For a a potential pointcut error to result in a failure, the error must transfer to a super-expression of the faulty pointcut expression, and from there to another super-expression, and so on, all the way to the most complex super-expression of the faulty pointcut expression, which is a pointcut expression on the right-hand side of some advice.*

A *super-expression* of a pointcut expression is a pointcut expression built up from the first pointcut expression.

Consider the following example:

```
pointcut p1() : call(public * void foo());
pointcut p2() : p1() && target(Bar);

before() : p2() {
   \\ some computation
}
```

The “nearest” super-expression of `call(public * void foo())` is `p1() && target(Bar)`. The most complex super-expression of both expressions is `p2()` on the right-hand side of the before advice. It is only when a fault in, say, $p1$, has an effect on $p2$, and

[2]still assuming that *if* pointcuts are side-effect free

further when the effect on $p2$ has an effect on the expression on the right-hand side of the advice, the fault can have any effect, since it is the expression at the advice that decides if the advice will execute, and that assigns value(s) to the advice's parameter(s) (if any).

The above necessary condition states that the fault in the simple pointcut expression e must transfer to e's super-expressions, i.e., to other pointcut expressions making us of e, such that at each step, another error occurs. The kind of error at each step does not matter, as long as it follows from the previous step and affects the next step. For instance, a *positive selection error* at one step could result in a *negative selection error* at the next step. At the time a pointcut fault f has transfered from the simplest pointcut expression containing f to the pointcut expression e' on the right-hand side of some advice, there is an error on a "higher level" than before. It is when a fault has transfered to e' that it is decided wether the advice will execute or not, and any advice formal parameters are bound to values that may be used inside the advice body. The error after evaluating e' can be *positive selection error*, *negative selection error* or *context exposure error*, as for any pointcut expression, but in terms of the RELAY model they are now *context errors*[3] rather than *potential errors*.

4.2.4.2 Propagation from Advice – Selection Errors

The conditions for a pointcut error to propagate further depends on the type of the error. A *positive selection error* after evaluating the pointcut expression at a piece of advice, means that the advice is decided to execute at a join point is was not intended to execute at. A *negative selection error* after evaluating the pointcut expression at a piece of advice, means that the advice is decided not to execute at a join point it was intended to execute at.

For both kinds of error, a minimum requirement for the error to possibly propagate to the output, is that the advice that is incorrectly decided to execute or not execute can have some effect on the program. For instance, executing some *after advice* after a join point cannot possibly affect

[3]not to be confused with *context exposure error*

execution if the advice body is empty. To be more precise, in the case of a *positive selection error*, the addition of advice a at a join point in program p has to result in a change in control- and/or data dependences compared to p *without* a. In the case of a *negative selection error*, the removal of a at a join point in p has to result in a change in control- and/or data dependences compared to p *with* a. The question then, is how the addition or removal of advice affects control- and data dependences.

First, consider *before* or *after* advice a in the case of a *positive selection error*. There is only one way a can change control dependences: by throwing exceptions. Advice a can contain a statement r that either explicitly throws an exception, or calls a method or constructor that may throw an exception. If a does not handle the exception with a try/catch construct, a thrown exception will propagated from a. For example, consider Figure 3.2(b), where control dependences are added from j to a, from $exit$ to a and from $ex.exit$ to a, compared to the CFG without a in Figure 3.1.

Before or after advice a can also add or remove existing data dependences in a program. For this to happen, a must contain a statement s that defines a variable v. But v must also be *used* by some statement s' in the program, following a. Additionally, there must be a path from s to s' that is definition-clear with respect to v. For example, consider Figure 3.8(b), where the definition of x in a, where a data dependence from s' to a is added, and a data dependence from s’ to s is removed, with the addition of advice a to the CFG in Figure 3.8(a).

Now, consider before advice a in the case of a *negative selection error*. For the removal of a to have an effect, the removal of a must result in the addition or the removal of some control- or data dependence. If a contained a statement that may throw an exception, the removal of a will result in the opposite change in control dependences that an addition of a would have. For example, consider the changes in data dependences from the CFG in Figure 3.2(b), to the CFG in Figure 3.1.

Just as for changes in control dependences, changes in data dependences when removing a are the opposite of the changes resulting from adding a. A propagation condition for before and after advice in the case of selection errors can now be formulated.

Pointcut Fault Propagation Condition 2. *For a positive selection error (or negative selection* *error) to cause a failure, where the unintended (or intended) advice a is before or after advice, a must contain a statement r that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in a, or that is a definition of a variable v that reaches a use of v in a statement r' following a.*

Next, consider *around advice* a in the case of a *positive selection error*, where a_1 is the part of a before a *proceed* statement, and a_2 is the part of a following the *proceed* statement. Without loss of generality, assume a contains at most one *proceed* statement. Around advice has the same means to change control- and data dependences as before or after advice, but because of the ability of *proceed* to execute the join point j "inside" a or to cancel j altogether, extra analysis is required.

An unconditional call to *proceed* in a does not affect existing control dependences; see Figure 3.4(a). Not including *any* call to *proceed* in a, however, removes existing control dependences, as j cannot possibly execute and is therefore not control dependent on any other node. For example, in Figure 3.1, j is control dependent on $start$, and in Figure 3.4(b), that control dependence is removed. Even if j was not control dependent on $start$ in the first place, it would be control dependent on *something*, e.g. on the decision to execute the program or not, but in the latter case j simply cannot execute and is not control dependent on anything. A *conditional* call to *proceed*, e.g. a *proceed* statement guarded by an *if* statement, also affects control dependences. For example, in Figure 3.4(c) a data dependence from j to a_1 is substituted for the data dependence from j to $start$ in Figure 3.1.

Since around advice a can execute j as part of a's own execution by calling *proceed*, there will be a changed data dependence when a is added to p, if, a_1 contains a statement r that defines a variable w, that is used by a statement r' in j, and there is a path from r to r' that is definition-clear with respect to w. The statement r' in j ends up being data dependent on r in a, instead of whatever statement (outside a) it was data dependent on before the addition of a.

The power of around advice a to cancel j altogether by not calling *proceed*, means that control- and data dependences can be affected in p even if a does not define variables or throws exceptions.

It is enough that j does so, since adding advice a may result in the removal of j from execution. Control dependences can be altered if j has a statement s that throws an exception or calls a method or constructor that may throw an exception, and there is no associated try/catch construct to handle that exception in j. A data dependence may be altered if s defines a variable x, there is a statement s' following j that uses x, and there is a path from s to s' that is definition-clear with respect to x.

The above observations on changed control- and data dependences when adding unintended around advice to a program (as is the case for a *positive selection error*), also apply when removing intended around advice from a program (which is the case for a *negative selection error*). This is true since if a control- or data dependence d is changed by adding advice a, d must necessarily also change when removing a.

A propagation condition for around advice in the case of selection errors can now be formulated.

Pointcut Fault Propagation Condition 3. For a *positive selection error* (or *negative selection error*) to cause a failure, where the unintended (or intended) advice a is *around* advice, with a_1 being the part of a before *proceed* and a_2 being the part of a after *proceed*, either

- a must contain a statement r that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in a, or that is a definition of a variable v that reaches a use of v in a statement r' following a
- a_1 must contain a statement s that is a definition of a variable w that reaches a use of w in a statement s' in j
- j must contain a statement t that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in j, or that is a definition of a variable x that reaches a use of x in a statement t' following j

4.2.4.3 Propagation from Advice – Context Exposure Error

When a pointcut error has propagated to a *context exposure error* in the pointcut expression on the right-hand side of some advice a, it also means that at least one parameter p of a contains an incorrect value. For this error to propagate further, a minimum requirement is that the advice is selected for execution.

Pointcut Fault Propagation Condition 4. *For a context exposure error in a parameter p in advice a to cause a failure, a must be selected to execute by a's pointcut expression.*

A *context exposure error* cannot affect control- or data dependences in a program like selection errors can. Rather, for a context exposure error to propagate, it must be used in a computation or a predicate statement such that the actual control flow and data flow is altered compared to what would have happened without the error.

As for selection errors, for a *context exposure error* to propagate certain constraints must hold about the syntactic structure of advice (all kinds of advice) or join point (around advice only).

Pointcut Fault Propagation Condition 5. For a *context exposure error* in parameter p of advice a to cause a failure, and a is *before* or *after* advice, a must contain a statement q that either

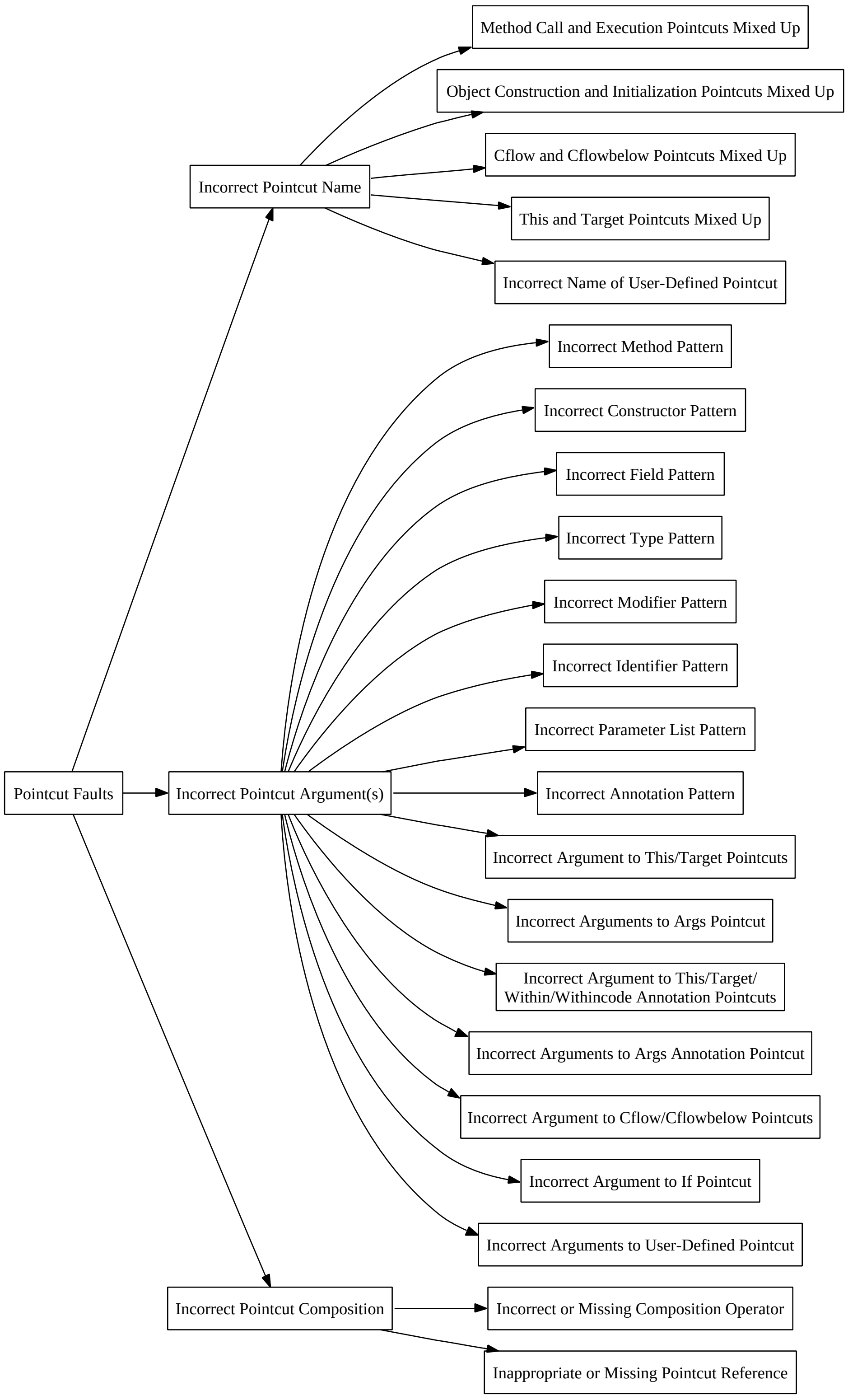
- is a computation using p, and the result of the computation is used by q to define a variable v, and the definition of v in q reaches a use of v in a statement q' following a
- is a predicate statement using p, and the evaluating q can result in the execution of a statement r that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in a, or is a definition of a variable w that reaches a use of w in a statement r' following a

Pointcut Fault Propagation Condition 6. For a *context exposure error* in parameter p of advice a to cause a failure, and a is *around* advice, where a_1 is the part of a before a call to *proceed*, a must contain a statement q that either

- is a computation using p, and the result of the computation is used by q to define a variable v, and the definition of v in q reaches a use of v in a statement q' following a
- is a predicate statement using p, and the evaluation of q may result in the execution of a statement r that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in a, or is a definition of a variable w that reaches a use of w in a statement r' following a
- is a predicate statement using p, and the evaluation of q may result in a call to *proceed*, and j contains a statement r that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in j, or is a definition of a variable x that reaches a use of x in a statement r' following j
- is a call to *proceed*, and p is an argument to proceed, which is bound to a formal parameter p' of j and j contains a statement s that either
  - is a computation using p', and the result of the computation is used by s to define a variable y, and the definition of y in s reaches a use of y in a statement s' following j
  - is a predicate statement using p', and the evaluation of s may result in the execution of a statement t that either throws an exception, calls a method or constructor that may throw an exception, and there is no associated catch block in j, or is a definition of a variable z that reaches a use of z in a statement t' following j

4.3 Pointcut Fault Types

This section describes a set of fault types that can appear in pointcuts. The fault types are classified as a hierarchy of categories, illustrated in Figure 4.1.



Pointcut Faults
Incorrect Pointcut Name
Method Call and Execution Pointcuts Mixed Up
Object Construction and Initialization Pointcuts Mixed Up
Cflow and Cflowbelow Pointcuts Mixed Up
This and Target Pointcuts Mixed Up
Incorrect Name of User-Defined Pointcut
Incorrect Pointcut Argument(s)
Incorrect Method Pattern
Incorrect Constructor Pattern
Incorrect Field Pattern
Incorrect Type Pattern
Incorrect Modifier Pattern
Incorrect Identifier Pattern
Incorrect Parameter List Pattern
Incorrect Annotation Pattern
Incorrect Argument to This/Target Pointcuts
Incorrect Arguments to Args Pointcut
Incorrect Argument to This/Target/ Within/Withincode Annotation Pointcuts
Incorrect Arguments to Args Annotation Pointcut
Incorrect Argument to Cflow/Cflowbelow Pointcuts
Incorrect Argument to If Pointcut
Incorrect Arguments to User-Defined Pointcut
Incorrect Pointcut Composition
Incorrect or Missing Composition Operator
Inappropriate or Missing Pointcut Reference

Figure 4.1: Pointcut fault categories.

4.3.1 Incorrect Pointcut Name

Faults in this category involve referring to another pointcut in a pointcut expression, but for some reason getting the name of the pointcut wrong. The pointcut may be user-defined or primitive. AspectJ includes a large number of primitive pointcut designators, and many of them have similar syntax and/or semantics. Using one primitive pointcut in a situation where another primitive pointcut should be used is therefore not unlikely. User-defined (i.e. named) pointcuts can also have similar syntax and/or semantics and make them equally prone to faults.

4.3.1.1 Method Call and Execution Pointcuts Mixed Up

This category involves using a *call* pointcut where an *execution* pointcut would be the correct choice, or vice versa, with a method pattern as argument. In addition to picking out the *calling* vs. the *called* side of a method, these pointcuts have subtle differences in semantics that especially manifest themselves in the context of declared-type patterns and inheritance. In addition, they both take the same argument, a method pattern, which makes the two pointcuts both semantically and syntactically close. It is assumed that the pointcut argument is correct.

Fault types in this category:

- **Call should be execution**
- **Execution should be call**

Example. Execution should be call.

```
pointcut p() : execution(int Foo.m());
```

should be

```
pointcut p() : call(int Foo.m());
```

Table 4.1 shows what kinds of errors the fault types in the category can result in.

Fault Type	**PSE**	**NSE**	**CEE**
Call should be execution	Yes	Yes	No
Execution should be call	Yes	Yes	No

Table 4.1: Fault types in the category Method Call and Execution Pointcuts Mixed Up and the errors instances of the faults can lead to. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

4.3.1.2 Object Construction and Initialization Pointcuts Mixed Up

This category involves capturing the creation of an object by using the incorrect pointcut for the task, i.e. mixing up the initialization, preinitialization, call, and execution pointcuts for a constructor. The faults types in this category are similar to the mix-up of call and execution for methods, but with the added complexity of two more pointcuts to choose from. The semantics of the execution, initialization and preinitialization pointcuts are especially close since they match join points *inside* the constructor and not join points on the calling side, as the *call* pointcut does. All four pointcuts take constructor signatures as arguments. It is assumed that the argument is correct.

Fault types in this category:

- **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**

Fault Type	**PSE**	**NSE**	**CEE**
Call should be execution	Yes	Yes	No
Execution should be call	Yes	Yes	No
Initialization should be preinitialization	Yes	Yes	No
Preinitialization should be initialization	Yes	Yes	No
Call should be initialization	Yes	Yes	No
Initialization should be call	Yes	Yes	No
Execution should be initialization	Yes	Yes	No
Initialization should be execution	Yes	Yes	No
Call should be preinitialization	Yes	Yes	No
Preinitialization should be call	Yes	Yes	No
Execution should be preinitialization	Yes	Yes	No
Preinitialization should be execution	Yes	Yes	No

Table 4.2: Fault types in the category Object Construction and Initialization Pointcuts Mixed Up and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Call should be preinitialization**
- **Preinitialization should be call**
- **Execution should be preinitialization**
- **Preinitialization should be execution**

Example. Execution should be initialization.

```
pointcut p() : execution(public Foo.new());
```

should be

```
pointcut p() : initialization(public Foo.new());
```

Table 4.2 shows what kinds of errors the fault types in the category can result in.

4.3.1.3 Cflow and Cflowbelow Pointcuts Mixed Up

This category involves capturing the dynamic scope in which a join point (given by the pointcut argument) is occurring, using the incorrect pointcut (i.e., cflow/cflowbelow). The cflow and cflowbelow pointcuts have very similar semantics, and choosing the right one in a given situation

Fault Type	**PSE**	**NSE**	**CEE**
Cflow should be cflowbelow	Yes	No	No
Cflowbelow should be cflow	No	Yes	No

Table 4.3: Fault types in the category Cflow and Cflowbelow Mixed Up and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

requires reasoning about the run-time behavior of the program. As they both take a pointcut as an argument they are also syntactically close. It is assumed that the pointcut argument (i.e., another pointcut) is correct.

Fault types in this category:

- **Cflow should be cflowbelow**
- **Cflowbelow should be cflow**

Example. Cflowbelow should be cflow.

```
pointcut p() : cflowbelow(call(int Bar.n(double,double)));
```

should be

```
pointcut p() : cflow(call(int Bar.n(double,double)));
```

Table 4.3 shows what kinds of errors the fault types in the category can result in.

4.3.1.4 This and Target Pointcuts Mixed Up

This category involves using the *this* pointcut where the *target* pointcut would be the correct choice, or vice versa. Both pointcuts match on dynamic types at the join point, and both take a single type or identifier as their argument. In many cases the two pointcuts will match and expose the same object, which may provide for many situations of coincidental correctness. It is assumed that the pointcut argument is correct.

Fault types in this category:

- **This should be target**

Fault Type	**PSE**	**NSE**	**CEE**
This should be target	Yes	Yes	Yes
Target should be this	Yes	Yes	Yes

Table 4.4: Fault types in the category This and Target Pointcuts Mixed Up and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Target should be this**

Example. This should be target.

```
pointcut p(String s) : call(int java.lang.String.length()) && this(s);
```

should be

```
pointcut p(String s) : call(int java.lang.String.length()) && target(s);
```

Table 4.4 shows what kinds of errors the fault types in the category can result in.

4.3.1.5 Incorrect Name of User-Defined Pointcut

User-defined (i.e., named) pointcuts can have similar syntax and/or semantics to other user-defined pointcut and make it easy to use the incorrect pointcut for the task at hand. It is assumed that the pointcut argument(s) are correct.

There is one fault type in this category:

- **Incorrect name of user-defined pointcut.** A user-defined pointcut is referenced in a pointcut expression, but the name of the pointcut identifies another pointcut than the intended one.

Example. Incorrect name of user-defined pointcut.

```
before() : accountWithdrawals() { // some computation}
```

should be

```
before() : accountActivities() { // some computation}
```

Table 4.5 shows what kinds of errors the fault type in the category can result in.

Fault Type	**PSE**	**NSE**	**CEE**
Incorrect name	Yes	Yes	Yes

Table 4.5: Fault types in the category Incorrect Name of User-Defined Pointcut and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

4.3.2 Incorrect Pointcut Argument(s)

A pattern is the argument to many pointcuts. They are used by the kinded pointcuts *call*, *execution*, *initialization*, *preinitialization*, *staticinitialization*, *get* and *set*, by the static scope pointcuts *within* and *withincode*, and indirectly by the *cflow* and *cflowbelow* pointcuts. Patterns are often built up from simpler patterns. For example, method patterns are built up from annotation patterns, modifier patterns, identifier patterns, argument list patterns and type patterns. In such cases, a fault could be the result of any of these subpatterns being incorrect. Other pointcuts, such as *this* and *target*, take types, identifiers, or wildcards as their arguments. The *cflow* and *cflowbelow* pointcuts take another pointcut as their argument, and the *adviceexecution* pointcut does not take any arguments at all.

4.3.2.1 Incorrect Method Pattern

A method pattern is a possible argument to a *call* or an *execution* pointcut. A method pattern consists of an optional annotation pattern, an optional modifier pattern, a return type pattern, an optional declaring type pattern, a method name pattern, an optional parameter list pattern, and an optional throws pattern:

[*AnnotationPattern*] [*ModifierPattern*] *ReturnTypePattern*

[*DeclaringTypePattern*.] *MethodNamePattern* ([*ParameterListPattern*])

[`throws` *ThrowsPattern*]

A fault in a method pattern can occur in any of these elements of the pattern. Annotation pattern faults are covered in Section 4.3.2.8, modifier pattern faults are covered in Section 4.3.2.5 and parameter list pattern faults are covered in Section 4.3.2.7. The return type pattern, declaring type pattern and throws pattern are all instances of the general type pattern, whose fault types are discussed in Section 4.3.2.4. A method name pattern is an instance of the more general identifier pattern, whose fault types are discussed in Section 4.3.2.6.

In addition, the category includes the following fault types.

- **Modifier pattern includes "abstract" together with "static", "final" or "synchronized".** In Java, a method declared *abstract* cannot also be declared *static, final* or *synchronized*, so such a pattern would not match any join points.
- **Modifier pattern includes "transient".** In Java, a method cannot be declared *transient*, so such a pattern would not match any join points.
- **Modifier pattern includes "volatile".** In Java, a method cannot be declared *volatile*, so such a pattern would not match any join points.
- **Declaring type pattern constitutes primitive type(s).** The type in which a method is declared cannot be a primitive type, so such a pattern would not match any join points.
- **Throws pattern does not constitute exception type(s).** Only subtypes of *java.lang.Throwable* can be declared in a throws clause, so such a pattern would not match any join points.

Example. Modifier pattern includes "abstract" together with "static", "final" or "synchronized".

```
pointcut p() : call(abstract synchronized * bar());
```

Table 4.6 shows what kinds of errors the fault types in the category can result in.

4.3.2.2 Incorrect Constructor Pattern

A constructor pattern can be the argument to the *call* or *execution* pointcuts, and is the only possible argument to an *initialization* or a *preinitialization* pointcut. A constructor pattern is similar to a method pattern, but lacks the return type pattern and has the keyword `new` instead of a method name pattern:

Fault Type	**PSE**	**NSE**	**CEE**
Modifier pattern includes abstract together with static, final or synchronized	No	Yes	No
Modifier pattern includes transient	No	Yes	No
Modifier pattern includes volatile	No	Yes	No
Declaring type pattern constitutes primitive type(s)	No	Yes	No
Throws pattern does not constitute exception type(s)	No	Yes	No

Table 4.6: Fault types in the category Incorrect Method Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

[*AnnotationPattern*] [*ModifierPattern*] [*DeclaringTypePattern.*]

`new` ([*ParameterListPattern*]) [`throws` *ThrowsPattern*]

Everything stated about the annotation pattern, modifier pattern, declaring type pattern, parameter list pattern and throws pattern in the previous section also apply to constructor patterns. However, there is no possibility of an incorrect return type pattern or method name pattern.

Fault types in this category:

- **Modifier pattern includes "abstract".** In Java, a constructor cannot be declared *abstract*, so such a pattern would not match any join points.

- **Modifier pattern includes "static".** In Java, a constructor cannot be declared *static*, so such a pattern would not match any join points.

- **Modifier pattern includes "final".** In Java, a constructor cannot be declared *final*, so such a pattern would not match any join points.

- **Modifier pattern includes "synchronized".** In Java, a constructor cannot be declared *synchronized*, so such a pattern would not match any join points.

- **Modifier pattern includes "transient".** In Java, a constructor cannot be declared *transient*, so such a pattern would not match any join points.

- **Modifier pattern includes "volatile".** In Java, a constructor cannot be declared *volatile*, so such a pattern would not match any join points.

Fault Type	**PSE**	**NSE**	**CEE**
Modifier pattern includes abstract	No	Yes	No
Modifier pattern includes static	No	Yes	No
Modifier pattern includes final	No	Yes	No
Modifier pattern includes transient	No	Yes	No
Modifier pattern includes volatile	No	Yes	No
Declaring type pattern constitutes interface(s) only	No	Yes	No
Declaring type pattern constitutes primitive type(s)	No	Yes	No
Attempting to use a method pattern to match constructors.	Yes	Yes	No

Table 4.7: Fault types in the category Incorrect Constructor Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Declaring type pattern constitutes interface(s) only.** The type in which a constructor is declared cannot be an interface, so such a pattern would not match any join points.
- **Declaring type pattern constitutes primitive type(s).** The type in which a constructor is declared cannot be a primitive type, so such a pattern would not match any join points.
- **Attempting to use a method pattern to match constructors.** In order to match a constructor, the keyword *new* must be used. An attempt to match *both* method names and constructors using the * wildcard would therefore not match constructors, only method names (a return type pattern would also need to be specified). In other words, if a return type pattern and an identifier pattern is specified in a pattern, the pattern is a method pattern and will only match methods.

Example. Attempting to use a method pattern to match constructors.

```
pointcut p() : call(public * Foo.*()); \\ will not match constructors
```

Table 4.7 shows what kinds of errors the fault types in the category can result in.

4.3.2.3 Incorrect Field Pattern

A field pattern is the argument to a *set* or a *get* pointcut, which match modification or read access of fields in a class. It is similar to the method and constructor patterns:

[*AnnotationPattern*] [*ModifierPattern*] [*FieldTypePattern*]

[*DeclaringTypePattern.*] *FieldNamePattern*

The annotation and modifier patterns have the same forms as in method and constructor patterns. Their fault types are discussed in Section 4.3.2.8 and Section 4.3.2.5 correspondingly. Both the field type pattern and the declaring type pattern are instances of the more general type pattern, whose fault types are discussed in Section 4.3.2.4. The field name pattern is an instance of the more general identifier pattern, discussed in Section 4.3.2.6.

Fault types in this category:

- **Modifier pattern includes "abstract".** In Java, a field cannot be declared *abstract*, so such a pattern would not match any join points.

- **Modifier pattern includes "synchronized".** In Java, a field cannot be declared *synchronized*, so such a pattern would not match any join points.

- **Modifier pattern includes both "final" and "volatile".** In Java, a field cannot be declared both *final* and *volatile*, so such a pattern would not match any join points.

- **Declaring type pattern constitutes interface(s) only.** The type in which a non-final field is declared cannot be an interface, and final fields are never matched [18, page 184], so such a pattern would not match any join points.

- **Declaring type pattern constitutes primitive type(s).** The type in which a field is declared cannot be a primitive type, so such a pattern would not match any join points.

Example. Declaring type pattern constitutes interface(s) only.

```
pointcut p() : get(int MyInterface.x);
```

Table 4.8 shows what kinds of errors the fault types in the category can result in.

Fault Type	PSE	NSE	CEE
Modifier pattern includes abstract	No	Yes	No
Modifier pattern includes synchronized	No	Yes	No
Modifier pattern includes both final and volatile	No	Yes	No
Declaring type pattern constitutes interface(s) only	No	Yes	No
Declaring type pattern constitutes primitive type(s)	No	Yes	No

Table 4.8: Fault types in the category Incorrect Field Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

4.3.2.4 Incorrect Type Pattern

Type patterns are used by every primitive pointcut that makes use of patterns. A type pattern can have an annotation pattern prefix, whose faults type are discussed in Section 4.3.2.8

Fault types in this category:

- **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**

Fault Type	**PSE**	**NSE**	**CEE**
Wildcard .. is used where * should be used	Yes	Yes	No
Wildcard * is used where .. should be used	Yes	Yes	No
Operator && is used between two types where || should be used	No	Yes	No
Operator || is used between two types where && should be used	Yes	No	No
Operator + is used after a type where it should not be used	Yes	No	No
Operator + is not used after a type where it should be used	No	Yes	No
Operator ! is used before a type where it should not be used	Yes	Yes	No
Operator ! is not used before a type where it should be used	Yes	Yes	No
Type is included that should not be included	Yes	Yes	No
Type is not included that should be included	Yes	Yes	No
Type(s) specified is (are) not visible in the scope of the pointcut	No	Yes	No
Mutually exclusive types are &&-ed together	Yes	No	No

Table 4.9: Fault types in the category Incorrect Type Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Type(s) specified is (are) not visible in the scope of the pointcut.** If one or more packages are not specified, only types visible from within the scope of the pointcut, directly or by using the Java `import` statement, will be matched.

- **Mutually exclusive types are "&&"-ed together.** For example, a type cannot be both *int* and *double*, so specifying *int && double* in a pattern would be a fault.

Example. Type(s) specified is (are) not visible in the scope of the pointcut.

```
// No import of java.sql.SQLPermission, and pointcut p is
// in another package.
pointcut p() : call(public SQLPermission.new(String));
```

Should have been e.g.:

```
import java.sql.*;
pointcut p() : call(public SQLPermission.new(String));
```

or:

```
pointcut p() : call(public java.sql.SQLPermission.new(String));
```

Table 4.9 shows what kinds of errors the fault types in the category can result in.

4.3.2.5 Incorrect Modifier Pattern

Modifier patterns are used in method patterns, constructor patterns and field patterns.

Fault types in this category:

- **Operator “!” is used before a modifier 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.** For example, a member cannot be both `public` and `private`, so specifying such a pattern would be a fault.
- **Modifier inappropriate for the pattern is included.** For example, a method cannot be declared `volatile`, so specifying such a pattern would be a fault. See sections 4.3.2.1, 4.3.2.2 and 4.3.2.3 for discussions of these faults in the different contexts.

Example. Operator “!” is used before a modifier where it should not.

```
// Will match only calls to methods not declared private
pointcut p() : call(!private void Foo.bar(..));
```

should have been

```
// Will match only calls to methods that are declared private
pointcut p() : call(private void Foo.bar(..));
```

Table 4.10 shows what kinds of errors the fault types in the category can result in.

4.3.2.6 Incorrect Identifier Pattern

Although the term *identifier* in Java has a broad meaning, for pointcut purposes it means a method name or a field name, since patterns for these two have the same form. No logic operators can be used in identifier patterns, and only the `*` wildcard can be used.

Fault types in this category:

Fault Type	**PSE**	**NSE**	**CEE**
Operator ! is used where it should not be used	Yes	Yes	No
Operator ! is not used before a modifier where it should be used	Yes	Yes	No
Modifier is included that should not be included	No	Yes	No
Modifier is not included that should be included	Yes	No	No
Mutually exclusive modifiers are included	No	Yes	No
Modifier inappropriate for the pattern is included	No	Yes	No

Table 4.10: Fault types in the category Incorrect Modifier Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

Fault Type	**PSE**	**NSE**	**CEE**
Wildcard * is used in a place where it should not be used	Yes	No	No
Wildcard * is not used in a place it should be used	No	Yes	No

Table 4.11: Fault types in the category Incorrect Identifier Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Wildcard “$*$” is used in a place where it should not be used**

- **Wildcard “$*$” is not used in a place where it should be used**

Example. Wildcard “$*$” is not used in a place where it should be.

```
// Will match calls to methods named "set"
pointcut p() : call(public void set(..));
```

should have been

```
// Will match calls to methods with name starting with "set"
pointcut p() : call(public void set*(..));
```

Table 4.11 shows what kinds of errors the fault types in the category can result in.

4.3.2.7 Incorrect Parameter List Pattern

Parameter list patterns are used in method patterns and constructor patterns. A parameter list pattern is a list of zero or more type patterns, so all fault types related to type patterns apply. In addition to individual type patterns, the wildcard .. can be used to specify any number of parameters.

Fault types in this category:

Fault Type	**PSE**	**NSE**	**CEE**
Incorrect number of parameters are listed	Yes	Yes	No
Order of parameters is incorrect	Yes	Yes	No
Parameter list pattern is empty when the wildcard .. should be used	No	Yes	No
Wildcard .. is used where a type pattern should be used	Yes	No	No

Table 4.12: Fault types in the category Incorrect Parameter List Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Incorrect number of parameters are listed.** If too many parameters are listed, or if one or more parameters are missing, the pattern will fail to match the desired set of parameter lists.
- **Order of parameters is incorrect.** The number and types of parameters might be correct, but the order specified fails to match the desired set of parameter lists.
- **Parameter list pattern is empty when the wildcard ".." should be used.** To match any parameter list, the pattern .. should be used. Providing an empty pattern will only match empty parameter lists.
- **Wildcard ".." is used where a type pattern should be used.** Using the wildcard .. between two commas does not match any type in that position of the list, but any number of parameters from that point on.

Example. Incorrect number of parameters are listed.

```
pointcut p() : call(public void foo(int, int));
```

should have been

```
pointcut p() : call(public void foo(int, int, int));
```

Table 4.12 shows what kinds of errors the fault types in the category can result in.

4.3.2.8 Incorrect Annotation Pattern

An annotation pattern can be used to match against the set of Java 1.5 metadata annotations on an annotated target, such as a method declaration or a class declaration. An annotation pattern

Fault Type	**PSE**	**NSE**	**CEE**
Operator ! is used before an element where it should not be used	Yes	Yes	No
Operator ! is not used before an element where it should be used	Yes	Yes	No
Element is included that should not be included	No	Yes	No
Element is not included that should be included	Yes	No	No

Table 4.13: Fault types in the category Incorrect Annotation Pattern and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

element has the following form @(*TypePattern*). Such simple elements may be negated using !, and combined by concatenation, e.g. !@Foo @Bar. Since an annotation pattern consists of one or more type patterns, all type pattern fault types apply to annotation patterns as well (see Section 4.3.2.4).

Fault types in this category:

- **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**

Example. Element is included that should not be included.

```
pointcut p() : get(@FooAnnotation @BarAnnotation private int MyClass.x);
```

should have been

```
pointcut p() : get(@FooAnnotation private int MyClass.x);
```

Table 4.13 shows what kinds of errors the fault types in the category can result in.

4.3.2.9 Incorrect Argument to This/Target Pointcuts

The parameter of the *this* and *target* pointcuts is a type, an identifier or the wildcard `*`. When a type is specified, join points are matched according to this type. If an identifier is specified it must be one of the pointcut or advice formal parameters, and join points are matched according to the type of the parameter. The wildcard `*` matches any type. Incorrect *this* and *target* pointcuts have the potential both for matching the incorrect set of join points and for incorrect exposure of context.

Fault types in this category:

- **Wildcard "$*$" should be type**
- **Wildcard "$*$" should be identifier**
- **Type should be wildcard "$*$"**
- **Identifier should be wildcard "$*$"**
- **Type should be indentifier**
- **Identifier should be type**
- **Type is incorrect type**
- **Identifier is the incorrect pointcut/advice parameter.** An identifier is specified that is one of the parameters in the parameter list of the pointcut/advice, but it is the incorrect parameter.

Example. Identifier is the incorrect pointcut/advice parameter.

```
pointcut p(Bar b1, Bar b2) : call(* foo()) && this(b1) && target(b2);
```

should have been

```
pointcut p(Bar b1, Bar b2) : call(* foo()) && this(b2) && target(b1);
```

Table 4.14 shows what kinds of errors the fault types in the category can result in.

Fault Type	PSE	NSE	CEE
Wildcard * should be type	Yes	No	No
Wildcard * should be identifier	Yes	No	No
Type should be *	No	Yes	No
Identifier should be wildcard *	No	Yes	Yes
Type should be identifier	Yes	Yes	No
Identifier should be type	Yes	Yes	Yes
Type is incorrect type	Yes	Yes	No
Identifier is the incorrect pointcut/advice parameter	Yes	Yes	Yes

Table 4.14: Fault types in the category Incorrect Argument to This/Target Pointcuts and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

4.3.2.10 Incorrect Arguments to Args Pointcut

The parameter of the *args* pointcut is a list of types, identifiers, and `*` wildcards, in addition to the wildcard `..`, which specifies any number of arguments. Each element in the list is therefore subject to the same fault types as the parameter of the *this* and *target* pointcuts, discussed in the previous section. An incorrect *args* pointcut has the potential both for matching the incorrect set of join points and for incorrect exposure of context.

Fault types in this category:

- **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 “..”**

Fault Type	**PSE**	**NSE**	**CEE**
Incorrect number of parameters is listed	Yes	No	No
Order of parameters is incorrect	Yes	Yes	Yes
Wildcard .. should be wildcard *	Yes	No	No
Wildcard * should be wildcard ..	No	Yes	No
Wildcard .. should be type	Yes	No	No
Wildcard .. should be identifier	Yes	Yes	No
Type should be wildcard ..	No	Yes	No
Identifier should be wildcard ..	No	Yes	Yes
Type should be identifier	Yes	Yes	No
Identifier should be type	Yes	Yes	Yes
Type is incorrect	Yes	Yes	No
Identifier is the incorrect pointcut/advice parameter	Yes	Yes	Yes

Table 4.15: Fault types in the category Incorrect Argument to Args Pointcut and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Identifier should be wildcard “..”**

- **Type should be indentifier**

- **Identifier should be type**

- **Type is incorrect.** The type should have been another type.

- **Identifier is the incorrect pointcut/advice parameter.** An identifier is specified that is one of the parameters in the parameter list of the pointcut/advice, but it is the incorrect parameter.

Example. Order of parameters is incorrect.

```
pointcut p() : call(* foo()) && args(Foo, Bar);
```

should have been

```
pointcut p() : call(* foo()) && args(Bar, Foo);
```

Table 4.15 shows what kinds of errors the fault types in the category can result in.

Fault Type	**PSE**	**NSE**	**CEE**
Annotation type should be identifier	Yes	Yes	No
Identifier should be annotation type	Yes	Yes	Yes
Annotation type is incorrect	Yes	Yes	No
Identifier is the incorrect pointcut/advice parameter	Yes	Yes	Yes

Table 4.16: Fault types in the category Incorrect Argument to This, Target, Within, Withincode, Annotation Annotation Pointcuts and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

4.3.2.11 Incorrect Argument to This, Target, Within, Withincode, Annotation Annotation Pointcuts

Faults in this category involves using the @this, @target, @within, @withincode, and @annotation pointcuts to select join points based on annotations, and expose context (i.e. annotation values) at the join points.

Fault types in this category:

- **Annotation type should be identifier**
- **Identifier should be annotation type**
- **Annotation type is incorrect.** The annotation type should have been another annotation type.
- **Identifier is the incorrect pointcut/advice parameter.** An identifier is specified that is one of the parameters in the parameter list of the pointcut/advice, but it is the incorrect parameter.

Example. Annotation type is incorrect.

```
pointcut p() : call(@FooAnnotation * foo());
```

should have been

```
pointcut p() : call(@BarAnnotation * foo());
```

Table 4.16 shows what kinds of errors the fault types in the category can result in.

4.3.2.12 Incorrect Arguments to Args Annotation Pointcut

Faults in this category involve using the @args pointcut to select join points based on annotations, and expose context (i.e. annotation values) at the join points. An argument to @args can be an annotation type or an identifier as for the other annotation pointcuts, but can also be the wildcard *. As for the regular args pointcut, the wildcard .. can be used to match any (annotation) type and number of arguments.

Fault types in this category:

- **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 “..”**

Example. Wildcard “..” should be wildcard “*”.

Fault Type	**PSE**	**NSE**	**CEE**
Incorrect number of parameters is listed	Yes	Yes	No
Order of parameters is incorrect	Yes	Yes	Yes
Wildcard .. should be wildcard *	Yes	No	No
Wildcard * should be wildcard ..	No	Yes	No
Wildcard * should be annotation type	Yes	No	No
Wildcard * should be identifier	Yes	No	No
Annotation type should be wildcard *	No	Yes	No
Identifier should be wildcard *	No	Yes	Yes
Wildcard .. should be annotation type	Yes	No	No
Wildcard .. should be identifier	Yes	No	No
Annotation type should be wildcard ..	No	Yes	No
Identifier should be wildcard ..	No	Yes	Yes

Table 4.17: Fault types in the category Incorrect Arguments to Args Annotation Pointcuts and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

```
pointcut p() : call(public void bar()) && @args(int, ..);
```

should have been

```
pointcut p() : call(public void bar()) && @args(int, *);
```

Table 4.17 shows what kinds of errors the fault types in the category can result in.

4.3.2.13 Incorrect Argument to Cflow/Cflowbelow Pointcuts

The *cflow* and *cflowbelow* pointcuts capture join points in the control flow of join points captured by another pointcut, given as the argument to cflow or cflowbelow. An incorrect argument to *cflow* or *cflowbelow* therefore means the argument is an incorrect pointcut expression. The pointcut expression might match the incorrect set of join points, leading to the *cflow* or *cflowbelow* pointcut matching the incorrect set of join points as well.

There is one fault type in this category.

- **Incorrect pointcut expression.** The incorrect pointcut expression can take many forms; in fact it can be an instance of any fault type described in this chapter.

Example. Incorrect pointcut expression.

Fault Type	**PSE**	**NSE**	**CEE**
Incorrect pointcut expression	Yes	Yes	No

Table 4.18: Fault types in the category Incorrect Argument to Cflow/Cflowbelow Pointcuts and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

```
pointcut p() : cflow(p1);
```

should have been

```
pointcut p() : cflow(p2);
```

Table 4.18 shows what kinds of errors the fault types in the category can result in.

4.3.2.14 Incorrect Argument to If Pointcut

The *if* pointcut has the following form:

```
if (BooleanExpression)
```

where *BooleanExpression* is any regular Java boolean expression. As such a faulty argument can take on any form syntactically permissible by the Java language.

Fault types in this category:

- **Incorrect boolean expression.** The expression is formulated such that it might evaluate incorrectly.
- **Expression has undesired side-effect.** The expression might evaluate correctly in all situations, but the expression has an undesired side-effect e.g. as the result of calling a method that itself has a side-effect. This type of fault does not result in any error defined by the fault/failure model, as if pointcut were considered side-effect free. It is nevertheless mentioned here for completeness.
- **Expression depends on side-effect of another pointcut.** The expression might be correct and not have any side-effects, but it relies on a side-effect of another if pointcut. While this

Fault Type	**PSE**	**NSE**	**CEE**
Incorrect boolean expression	Yes	Yes	No
Expression has undesired side effect	N/A	N/A	N/A
Expression depends on side-effect of another pointcut	Yes	Yes	No

Table 4.19: Fault types in the category Incorrect Argument to If Pointcut and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

is not necessarily a fault, it is an anomaly, since evaluation order of pointcuts is undefined and side-effects of other if pointcuts should not be relied on.

Example. Incorrect boolean expression.

```
pointcut p() : if(x == 5);
```

should have been

```
pointcut p() : if(y == 5);
```

Table 4.19 shows what kinds of errors the fault types in the category can result in.

4.3.2.15 Incorrect Argument to User-Defined Pointcut

The previous sections have described incorrect arguments to primitive pointcuts. However, user-defined pointcuts might also take arguments, and they might be specified incorrectly by the pointcut expression that uses them. For a user-defined pointcut it does not matter if an argument is a type or the wildcard "*", as the type does not affect join point selection. If pointcut p_1 uses a user-defined pointcut p_2 and provides a type or "*" as the parameter, it does so simply because it does not need the context provided by p_2. It is assumed that the name of the pointcut is correct (and thus the incorrect number of arguments cannot be specified).

There is one fault type in this category:

- **Order of parameters is incorrect**

Example. Order of parameters is incorrect.

Fault Type	PSE	NSE	CEE
Order of parameters is incorrect	Yes	Yes	No

Table 4.20: Fault types in the category Incorrect Argument to User-Defined Pointcut and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

```
// Advice need only the target object, but incorrectly gets the
// current object instead
pointcut p(String s1, String s2) : this(s1) && target(s2);
before(String s) : p(s, *) {
   \\ some computation
}
```

should have been

```
pointcut p(String s1, String s2) : this(s1) && target(s2);
before(String s) : p(*, s) {
   \\ some computation
}
```

Table 4.20 shows what kinds of errors the fault types in the category can result in.

4.3.3 Incorrect Pointcut Composition

Pointcut composition involves creating more complex pointcuts from simpler ones. The simpler pointcuts might themselves be complex pointcuts, or simple pointcuts like a *call* primitive pointcut. The individual pointcuts in an expression might be correct, but if they are combined in an incorrect way we can still get incorrect results.

4.3.3.1 Incorrect or Missing Composition Operator

Pointcuts in a pointcut expression can be combined with the logic operators `!`, && and `||`.

Fault types in this category:

- **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**

Fault Type	**PSE**	**NSE**	**CEE**
Operator || is used between two pointcuts where && should be used	Yes	No	No
Operator && is used between two pointcuts where || should be used	No	Yes	No
Operator ! is used before a pointcut where it should not be used	Yes	Yes	No
Operator ! is not used before a pointcut where it should be used	Yes	Yes	No

Table 4.21: Fault types in the category Incorrect or Missing Composition Operator and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

- **Operator “!” is not used before a pointcut where it should be used**

Example. Operator “!” is not used before a pointcut where it should be used.

```
pointcut p() : call(int foo())
               && cflowbelow(execution(public void bar()));
```

should have been

```
pointcut p() : call(int foo())
               && !cflowbelow(execution(public void bar()));
```

Table 4.21 shows what kinds of errors the fault types in the category can result in.

4.3.3.2 Inappropriate or Missing Pointcut Reference

Faults in this category involve which other pointcuts are specified or should be specified when composing a more complex pointcut from simpler ones.

Fault types in this category:

- **Pointcut is referenced that should not be referenced.** A pointcut expression includes the name of a pointcut that should not be included.

- **Pointcut that should be referenced is not referenced.** A pointcut expression does not include the name of a pointcut that should be included.

Example. Pointcut is referenced that should not be referenced.

```
pointcut p() : call(int foo()) && myPointcut();
```

Fault Type	**PSE**	**NSE**	**CEE**
Pointcut is referenced that should not be referenced	Yes	Yes	No
Pointcut that should be referenced is not referenced	Yes	Yes	No

Table 4.22: Fault types in the category Inappropriate or Missing Pointcut Reference and the errors they can result in. PSE = Positive Selection Error, NSE = Negative Selection Error, CEE = Context Exposure Error.

should have been

```
pointcut p() : call(int foo());
```

Table 4.22 shows what kinds of errors the fault types in the category can result in.

CHAPTER FIVE

ADVICE FAULTS

5.1 Introduction

This chapter presents a set of fault types that can occur in advice, and investigates their possible effects on program state.

5.2 Fault/Failure Model for Advice

The types of faults that can occur in advice are more diverse in their forms and effects than those that can appear in pointcuts, so the fault/failure model is interpreted individually for each type of fault and/or category. The reason for this difference is that while every pointcut has at most two tasks, to select a join point or not, and to expose context, the constructs associated with advice are used for several purposes. There is however one condition that must be true for any advice fault to execute.

Advice Fault Execution Condition. *For an advice fault to execute, some join point must occur that is selected by the pointcut expression on the right-hand side of the advice.*

As for pointcut faults, for a fault to cause an infection and for that infection to propagate, control flow and/or data flow must be altered. Most of the fault types in this chapter can alter control and/or data dependences of a program. Other faults cannot alter the dependences, but may alter the actual flow through the program.

For all kinds of faults, for an error to propagate to the observable output and cause a failure, there must be a chain of control- and/or data dependences from the output statement that caused the failure, back to the statement where the infection occurred.

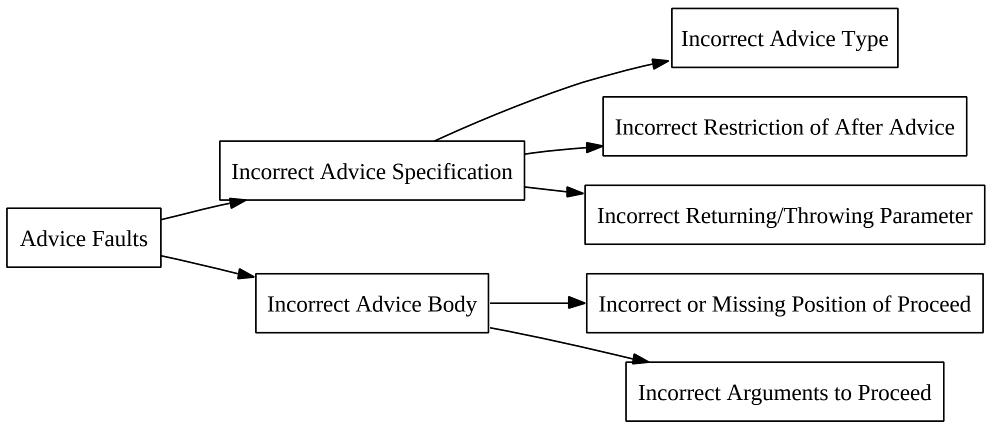

Advice Faults
Incorrect Advice Specification
Incorrect Advice Body
Incorrect Advice Type
Incorrect Restriction of After Advice
Incorrect Returning/Throwing Parameter
Incorrect or Missing Position of Proceed
Incorrect Arguments to Proceed

Figure 5.1: Advice fault categories.

5.3 Advice Fault Types

This section describes the categories and individual fault types that can occur in advice. The categories are depicted in Figure 5.1.

5.3.1 Incorrect Advice Specification

Faults in this category means that the specification part, i.e., the *AdviceSpec* in the definition of some advice is incorrect:

[`strictfp`] *AdviceSpec* [`throws` *TypeList*] `:` *Pointcut expression* { *Body* }

where *AdviceSpec* is one of

`before (` *Formals* `)`

`after (` *Formals* `) returning` [`(` *Formal* `)`]

`after (` *Formals* `) throwing` [`(` *Formal* `)`]

`after (` *Formals* `)`

Type `around (` *Formals* `)`

The advice specification specifies the type of advice (*before*, *after*, *around*) parameters to the advice (*before (Parameters)*, *after (Parameters)*, *around (Parameters)*) and the returning and throwing clauses (*returning (Parameter)*, *throwing (Parameter)*), the return type of a piece of around advice, and the restriction of *after* advice (*returning*, *throwing*, none).

5.3.1.1 Incorrect Advice Type

Faults in this category result in a piece of advice being executed in the incorrect position relative to the join point, e.g. *before* the join point instead of *after* it. However, certain constraints must hold about the syntactic structure of a piece of advice and/or the join point in order for a fault in this category to cause a change in control- or data dependences, which is required in order to cause an infection.

It is assumed that *around* advice do not contain *proceed* statements, since if they did, they could not erroneously be *before* or *after* advice.

Fault types in this category:

- ***Before* should be *after*.** Infection condition: For a fault of this type to cause an infection, in the *incorrect* program p, the advice a must contain a statement s that either
  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement s' in j
  - is a use of a variable w that is defined by a statement s' in j, and the definition of w in s' reaches the use of w in s in the correct program p'.
- ***After* should be *before*.** Infection condition: For a fault of this type to cause an infection, in the *incorrect* program p, the advice a must contain a statement s that either
  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement s' in j in the correct program p'

  - is a use of a variable w that is defined by a statement s' in j, and the definition of w in s' reaches the use of w in s

- ***Before* should be *around*.** Infection condition: For a fault of this type to cause an infection, in the *incorrect* program p, the join point j must contain a statement s that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in j
  - is a definition of a variable v that reaches a use of v in a statement s' following j

- ***Around* should be *before*.** *Infection condition:* For a fault of this type to cause an infection, in the *correct* program p', the join point j must contain a statement that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in j
  - is a definition of a variable v that reaches a use of v in a statement s' following j

- ***After* should be *around*.** Infection condition: For a fault of this type to cause an infection, in the *incorrect* program p, the join point j must contain a statement s that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in j
  - is a definition of a variable v that reaches a use of v in a statement s' following j

- ***Around* should be *after*.** Infection condition: For a fault of this type to cause an infection, in the *correct* program p', the join point j must contain a statement s that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in j

    - is a definition of a variable v that reaches a use of v in a statement s' following j

Example. Before should be *after*.

```
before() : myPointcut() {
   // ...
   if (x == 5) {
      throw new Exception(); // will result in bypassing join point
   }
   // ...
}
```

should have been

```
after() : myPointcut() {
   // ...
   if (x == 5) {
      throw new Exception(): // will not result in bypassing join point
   }
   // ...
}
```

5.3.1.2 Incorrect Restriction of After Advice

After returning advice executes after a join point only if the join point returned normally. *After throwing* advice executes after a join point only if the join point returned with an exception. Regular *after* advice executes after a join point in both cases. Faults in this category affects in which situations a piece of *after* advice a executes. Sometimes a might execute when a should not, other times a might not execute when a should.

Fault types in this category:

- ***After* should be *after returning*.** Infection condition: For a fault of this type to cause an infection, the join point j must include a statement s that either throws an exception, or calls a method or constructor that may throw an exception, and there is no associated catch block in j, *and* the advice a must contain a statement t that either
    - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a

  - is a definition of a variable v that reaches a use of v in a statement t' following a

- ***After returning* should be *after*.** Infection condition: As for *after should be after returning*.

- ***After* should be *after throwing*.** Infection condition: For a fault of this type to cause an infection, the join point j must include a path p through j in which an exception is not guaranteed to be thrown, *and* the advice a must include a statement t that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement t' following a

- ***After throwing* should be *after*.** Infection condition: As for *after should be after throwing*.

- ***After returning* should be *after throwing*.** Infection condition: For a fault of this type to cause an infection, the advice a must contain a statement s that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement s' following a

- ***After throwing* should be *after returning*.** Infection condition: As for *after throwing should be after returning*.

Example. After returning should be *after*.

```
// Code corresponding to join point, i.e., a method execution
void foo(String s) {
   // ...
   if (s.equals("bar")) {
      throw new BarException();
   }
   // ...
}
// Advice - will not run if join point throws exception
after() returning() : execution(void foo(String)) {
   // ...
}
```

should have been

```
// Code corresponding to join point, i.e., a method execution
void foo(String s) {
   // ...
   if (s.equals("bar")) {
      throw new BarException();
   }
   // ...
}
// Advice - will run if join point throws exception
after() : execution(void foo(String)) {
   // ...
}
```

5.3.1.3 Incorrect Returning/Throwing Parameter

Faults in this category result in advice executing at join points they should not, or advice not executing at join points they should, since the type of a returning or throwing parameter further restricts the join points that the advice should run at.

For fault types in this category, a common infection condition can be stated. For a fault in this category to cause an infection, there must be a path through the join point j in which an exception is not guaranteed to be thrown, and the advice a must contain a statement s that either

- throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
- is a definition of a variable v that reaches a use of v in a statement s' following a

Fault types in this category:

- **Returning parameter is specified but should not be specified.** Only join points returning with a value of the specified type will result in the advice being executed, rather than join points returning with a value of any type.

- **Returning parameter is not specified but should be specified.** Join point returning with a value of any type will result in the advice being executed, rather than only join points returning with a value of a specified type.

- **Returning parameter has incorrect type.** Depending on the parameter's type, advice a might execute at join points a was not intended to execute at, and not execute at join points a was intended to execute at.

- **Throwing parameter is specified but should not be specified.** Only join points throwing an exception of the specified type will result in the advice being executed, rather than join points throwing (or propagating) an exception of any type.

- **Throwing parameter is not specified but should be specified.** Join point throwing an exception of any type will result in the advice being executed, rather than only join points throwing an exception of a specified type.

- **Throwing parameter has incorrect type.** Depending on the parameter's type, advice a might execute at join points a was not intended to execute at, and not execute at join points a was intended to execute at.

Example. Throwing parameter is specified but should not be specified.

```
after() throwing(IOException e) : myPointcut() {
   // ...
}
```

should have been

```
after() throwing() : myPointcut() {
   // ...
}
```

5.3.2 Incorrect Advice Body

Faults in this category involve the body of a piece of advice, i.e., the *Body* part of an advice definition:

[`strictfp`] *AdviceSpec* [`throws` *TypeList*] `:` *Pointcut expression* { *Body* }

The only difference between the body of advice and regular Java methods is the possible presence of a *proceed* statement, so the fault types in this category all involve the use of this statement.

5.3.2.1 Missing or Incorrect Position of Proceed

Fault types in this category:

- **Advice has a *proceed* statement but should not.** *Infection condition:* For a fault of this type to cause an infection, in the *incorrect* program p, the join point j must contain a statement s that either
  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement s' following j
- **Advice does not have a *proceed* statement but should.** *Infection condition:* For a fault of this type to cause an infection, in the *correct* program p', the join point j must contain a statement s that either
  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement s' following j
- ***Proceed* statement is guarded by condition it should not be guarded by.** *Infection condition:* For a fault of this type to cause an infection, the join point j must contain a statement s that either

  - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in a
  - is a definition of a variable v that reaches a use of v in a statement s' following j

- ***Proceed* statement is not guarded by condition it should be guarded by.** *Infection condition:* As for *proceed statement is guarded by condition it should not be guarded by*.

Example. Proceed statement is guarded by condition it should not be guarded by.

```
void around(String s) : myPointcut(s) {
   // ...
   if (!s.length == 0) {    // incorrect - proceed should have been
      proceed(s);           // called in any case
   }
   // ...
}
```

5.3.2.2 Incorrect Argument(s) to Proceed

There is one fault type in this category:

- **Argument to proceed has incorrect value.** *Infection condition:* For a fault of this type to cause an infection, the formal parameter q of j bound to the argument r of *proceed* must reach a use of q in a statement s' in j, and s' is also either
  - a definition of a variable v that reaches a use of v in a statement s'' following j
  - a conditional statement that results in the execution of a statement t that either
    - throws an exception or calls a method or constructor that may throw an exception, and there is no associated catch block in j
    - is a definition of a variable w that reaches a use of w in a statement t' following j

Example.

```
// Advice
void around(String s) : myPointcut(s) {
   // ...
   proceed("argument:"); // incorrect, should have been, say,
                         // "argument: " + s
   // ...
}

// Code corresponding to join point, i.e., a method execution
void foo(String s) {
   // ...
   // since s is incorrect, condition might evaluate incorrectly:
   if (s.length() < 10) {
      throw new IncorrectArgumentException();
   }
   // ...
}
```

CHAPTER SIX

DISCUSSION

6.1 Introduction

This chapter discusses contributions and limitations of the proposed fault model.

We believe that the fault model relatively directly can be used to evaluate existing testing strategies using fault seeding, to create mutation operators for mutation testing, and to create code inspection checklists. An approach for evaluating testing strategies is outlined, example mutation operators are defined, and an example code inspection checklist is presented.

We also believe that the fault model can be used to derive test adequacy criteria. Ideas for criteria is outlined, including variants of mutation adequacy, branch and conditional coverage, and definition-use association coverage. More research is needed to come up with good, concrete adequacy criteria. Once test adequacy criteria have been developed, associated testing strategies can be devised.

Certain language features are not covered by the fault model. The third important construct of AspectJ next to pointcuts and advice, *inter-type declarations*, are not discussed, and neither is the notion of *advice precedence*, which decides the execution order of advice when there is more than one piece of advice to run at a single join point. This is obviously an argument against the usefulness of the model.

Another important limitation is that the fault model has not been empirically evaluated. Because of this, we do not know for sure if the faults described in the fault model are the kinds of faults that are likely to appear in real programs.

6.2 Contributions

6.2.1 Fault Seeding

Fault seeding (also called *error seeding*), is a technique originally proposed to estimate the number of faults that remain in software [62]. By this method, artificial faults are introduced into the program under test in some suitable random fashion unknown to the tester. It is assumed that these artificial faults are representative of the inherent faults in the program in terms of difficulty of detection. Then, the program is tested and the inherent and artificial faults discovered are counted individually. Let r be the ratio of the number of artificial faults found to the number of total artificial faults. Then the number of inherent faults in the program is statistically predicted with a maximum likelihood to be f/r, where f is the number of inherent faults found by testing. The method can also be used to measure the quality of a testing approach (which is proposed in the next section). The ratio r of the number of artificial faults found to the total number of artificial faults can be considered a measure of the test adequacy.

The fault model presented in this thesis can provide the artificial faults to be seeded into an AspectJ program. With fault seeding it is important that the difficulty of detection of the artificial faults are comparable to the inherent faults. While the fault model does not provide such an analysis, it is believed to reflect real faults made by programmers. If an artificial fault is of the same type as an inherent fault, it can be assumed that it is also comparably difficult to detect.

6.2.2 Evaluation of Testing Strategies

A measure of the quality of a (fault-directed) testing strategy is its ability to detect faults. Several papers in the literature proposing testing strategies for aspect-oriented programs in the literature have also reported on preliminary studies on how well the strategy detects certain kinds of faults. The evaluation has mainly been against the fault model proposed by Alexander et al. [9]. For instance, Xu and Xu [53, 58] argued that their testing approach will help detect pointcuts picking up extra join points, pointcuts missing join points, incorrect advice types and incorrect advice implementation. Lemos et al. [32] argued that their testing criteria could discover unintended selected join points, missing intended join point and incorrect advice execution order. Naqvi et al. [42] did an informal comparison of three testing strategies also using the fault model of Alexander et al.

This thesis provides a comprehensive fault model that can be a fundament for evaluation of testing strategies. An study comparing the quality of several testing strategies could for example follow an outline like this:

1. Develop a program with AspectJ

2. Seed faults into the program that are instances of fault types described in the fault model

3. Test the program with the various testing strategies and record the results

4. Evaluate to what extent the different strategies detected the seeded faults.

Under the assumption that the fault model describes faults that are actually likely to be present in AspectJ programs, such a study can be a good indicator of the quality of different testing approaches. Being able to do such evaluations is important, since the number of people developing AspectJ programs is increasing, and so is the number of available testing strategies. While other factors than fault detecting capability come into play in selecting a strategy, a certain ability to detect faults is a minimum requirement for any good testing strategy.

6.2.3 Mutation Testing

Mutation testing [20] is a more systematic approach to fault seeding, and was proposed as a procedure for evaluating the degree to which a program has been tested, that is, to measure test adequacy. Assume we have a program p and a test set t that has been generated in some fashion. The first step in mutation analysis is the construction of a collection of alternative programs that differ from the original program in some fashion. These alternatives are called *mutants* of the original program

[62]. Each mutant is then executed on each member of the test set t, stopping either when an element of t is found on which p and the mutant program produces different outputs, or when there are no more tests in t. In the former case we say that the mutant has *died*, since it is of no further value, whereas in the latter case we say the mutant *lives*. A mutant can be alive for one of two reasons; the test data are inadequate, or the mutant is equivalent to the original program [62]. If a large proportion of mutants live, we have no more reason to believe that p is correct than to believe that any of the live mutants are correct. If the test data are inadequate, the procedure is to generate more tests in order to *kill* the remaining mutants. A *mutation operator* is a syntactic transformation that produces a mutant when applied to the program under test [62]. It applies to a certain syntactic structure in the program and replaces it with another. Typically, mutation operators are designed on the basis of typical programmer errors.

We believe that the proposed fault model is a good source for creating mutation operators for AspectJ programs. Since the fault model is based on the competent programmer hypothesis [19], all the described fault types represent small syntactic changes to a correct program. Thus, it is relatively straightforward to turn a fault type into a mutation operator.

For instance, consider the fault category *object construction and initialization pointcuts mixed up*, described in Section 4.3.1.2. Table 6.1 shows each fault type and a corresponding mutation operator. To use the mutation operator "replace 'call' with 'execution"', for instance, a pointcut expression in the program is found that is a *call* pointcut, and 'call' is transformed into 'execution', with the argument to the pointcut intact (since both *call* and *execution* take a constructor pattern). The transformation could be manual, but should be automated to make the approach practically feasible.

Mutation operators can also be used on advice, e.g. replacing 'before' with 'after' or removing 'proceed' from around advice.

Additionally, the fault/failure analysis of each fault type and/or category can help in detecting

Fault Type	**Mutation Operator**
Call should be execution	Replace ‘execution’ with ‘call’
Execution should be call	Replace ‘call’ with ‘execution’
Initialization should be preinitialization	Replace ‘preinitialization’ with ‘initialization’
Preinitialization should be initialization	Replace ‘initialization’ with ‘preinitialization’
Call should be initialization	Replace ‘initialization’ with ‘call’
Initialization should be call	Replace ‘call’ with initialization’
Execution should be initialization	Replace ‘initialization’ with ‘execution’
Initialization should be execution	Replace ‘execution’ with ‘initialization’
Call should be preinitialization	Replace ‘preinitialization’ with ‘call’
Preinitialization should be call	Replace ‘call’ with ‘preinitialization’
Execution should be preinitialization	Replace ‘preinitialization’ with ‘execution’
Preinitialization should be execution	Replace ‘execution’ with ‘preinitialization’

Table 6.1: Example mutation operators.

equivalent mutants. The fault/failure analysis provides constraints that must hold about the syntactic structure of a program in order for a fault of a specific type or category to cause an infection and propagate to the output. If a certain mutation operator is used, and the fault type that it reflects cannot cause an infection and propagation given the structure of the program under test, then the mutant necessarily is equivalent to the original program.

6.2.4 Program Inspection

Program inspections are reviews of program code whose objective is the detection of faults. The notion of a formalized inspection process was first developed at IBM in the 1970s and was described by Fagan [22]. It is now a widely used method of program verification [45]. Program inspection is carried out by a small team of at least four people, first individually and then during an inspection meeting. A key to the inspection process is a checklist of common programmer errors. Again under the assumption that the fault model in this thesis reflects common programmer errors, it can be a basis for creating such checklists.

An simple checklist for inspection of advice could for example look like the following:

- If the advice has formal parameters, is each bound correctly in the pointcut expression?

- Is the type of the advice correct? (before/after/around)?
- For after advice, is the correct restriction used (returning/throwing/none)?
- For after returning advice, is the parameter specified/not specified correct?
- For after throwing advice, is the parameter specified/not specified correct?
- For around advice, if there is no proceed statement, is this correct?
- For around advice, for each proceed statement, will it be reached in all situations where it should?
- For around advice, for each proceed statement, is the argument correct?

6.2.5 Test Adequacy Criteria

Test adequacy criteria are criteria that tells a tester whether a program has been adequately tested, or to what extent it has been adequately tested [62]. Well-known test adequacy criteria are *statement coverage*, that requires that all the statements in the program under test are executed during testing, *branch coverage*, that requires all the control transfers in the program under under test to be exercised, and *all conditions coverage*, that requires every combination of truth values in the atomic predicates of all conditions in the program under test to be covered. *Mutation adequacy* is also a criterion, that measures the percentage of dead mutants compared to the mutants that are not equivalent to the original program [62].

An important question is what adequacy criteria are needed for AspectJ programs. *Pointcut expressions* are in many ways similar to conditional statements (e.g. *if* statements) in traditional programs. Aside from exposing context, a pointcut has two possible outcomes at a join point: select or not select. Pointcut expressions operate on sets (of join points), while conditional statements operate on Boolean expressions, which are similar. Because of these similarities, forms of branch coverage and condition coverage have been proposed in the literature as adequacy criteria for pointcuts [17, 50]. Mutation adequacy has also be proposed [40, 31]. We are in the view that both branch/conditional coverage and mutation adequacy are strong candidates for adequacy criteria for AspectJ programs. Mutation adequacy seems like the easiest choice, and an approach to mutation testing based on the fault model was outlined in Section 6.2.3. Mutation adequacy can be used for both pointcuts and advice.

The *pointcut fault execution condition* in Chapter 4 states that *a pointcut fault is executed if and only if the simplest pointcut expression containing that fault is evaluated.* This means that in order to be sure that a fault in a pointcut expression is executed, in testing it we should exercise each individual simple expression. If a faulty expression is not executed under test, there is no way that fault can be detected. Our analysis shows that this might be a difficult goal to achieve. One reason is that of *controllability*. In contrast to conditional statements in a traditional program, we cannot force a pointcut to be run by inputting appropriate test data. If and when a pointcut is evaluated is left undefined by the language [4, 27] and might happen at weave-time or at run-time. The other problem is that of *observability*. We have no direct way of observing the outcome of a pointcut evaluation. The only observation we can make is the execution of advice if the pointcut expression at the advice chooses to select a join point.

In the current AspectJ implementations [1, 5], pointcuts do not exist at run-time, i.e., pointcuts that can be evaluated statically are evaluated at compile- or load time, while pointcuts that need dynamic evaluation is transformed into conditional statements in the woven code. An approach is therefore to test pointcuts *indirectly* by exploiting how a particular language implementation works. While this may not be the most desirable approach, it is still valuable if it can lead to better detection of faults. Pointcuts requiring dynamic evaluation can then be tested at run-time, e.g. using regular branch coverage on byte code, while pointcuts that can be evaluated statically must be “tested” either at compile- or load time. Recompiling a program for each new test case is obviously undesirable; load-time weaving would be a slightly more viable approach.

Adequacy criteria for *advice* can also be derived from the fault model. Faults in the category *incorrect advice type* described in Section 5.3.1.1 (e.g., using 'before' instead of 'after'), are for example sensitive to the occurrence of exceptions thrown by the advice and definition-use pairs between the advice and the join point. Natural adequacy criteria would therefore be to require all *throw* statements to be executed and some form of definition-use coverage [62]. For around advice with a conditional call to *proceed*, a possible criterion would be to require both paths that *included* and paths that *did not include* the *proceed* statement, to be exercised.

In summary, we believe that the fault model can be a useful guide in devising test adequacy criteria, but further research is needed in order to come up with good, concrete criteria.

6.3 Limitations

6.3.1 Language Features not Covered

The proposed fault model covers most features associated with pointcuts and advice. There are two exceptions. *If* pointcuts are assumed side-effect free, and *advice precedence* is not covered. Not considering side-effects or *if* pointcuts is probably not a big limitation, since Boolean expressions seldom have side-effects, and letting them have so is by many considered bad programming practice. Nevertheless, side-effect related faults can exist, and their exclusion from the fault model is a weakness. A more serious limitation is the fact that advice precedence is not covered. The notion of advice precedence decides the order in which each piece of advice executes in the case of several advice woven at the same join point. Precedence also determines how one advice affect another, when using *proceed* and when throwing exceptions. The rules for advice precedence are quite complex, and are arguably a source of subtle faults in AspectJ programs.

Certain language features besides pointcuts and advice are not addressed. The most important of these are *inter-type declarations,* which allow a programmer to affect the static structure of a program by changing the inheritance hierarchy, introducing methods and fields into classes, turn checked exceptions into unchecked exceptions ("exception softening") etc. Inter-type declarations is considered a secondary feature to pointcuts and advice, but should be covered by a complete fault model. Other language features not covered include *aspect instantiation*, *aspect inheritance* and *aspect precedence*.

6.3.2 Empirical Evaluation

The fault model was conceived by carefully analyzing each language construct in terms of its syntax and semantics, and looking for problematic aspects of each construct — if a programmer was to use this feature, what would be likely mistakes? The author's personal experience with both AspectJ and other languages, including Java, C and C++, helped in this process. As a supplement, problematic issues raised by other researchers and practitioners were considered. An example of this is the fault category *method call and execution pointcuts mixed up*. The exact semantic differences between the *call* and *execution* pointcuts are widely considered difficult to understand, and have been discussed extensively both on the *aspectj-users* mailing list[1] and in the literature [13]. Textbooks and tutorials on AspectJ have also been examined in search for common mistakes and pitfalls [30, 18, 4, 2].

However, the fault model is not the result of an empirical study, neither has it been empirically evaluated, so we cannot know for sure that the proposed fault types are the faults that are actually most likely to appear in AspectJ programs. Some faults described may not bee likely faults at all, while some likely real faults are not covered by the model. Our hope is that a subset of significant size of all likely faults has been included. This is not to say that the fault model is without value, just that a next step of empirical evaluation is needed.

6.3.3 Formal Analysis

The fault/failure analysis in this thesis is based on common sense and informal definitions of the AspectJ language. It is not a rigorous mathematical analysis. A formal language model and a formal analysis is needed if one wishes to prove the statements and conditions stated about execution, infection and propagation. A formal treatment might moreover give further insights, such as

[1]https://dev.eclipse.org/mailman/listinfo/aspectj-users/

stronger conditions. The lack of formal analysis also means that there is a possibility of the stated conditions being imprecise or incorrect. It should be noted however, that a formal analysis might not be practically feasible because of the lack of an official and precise language specification.

CHAPTER SEVEN

CONCLUSIONS AND FUTURE WORK

This thesis has presented a fault model for pointcuts and advice in AspectJ Programs. Pointcuts and advice are the two principal features of AspectJ, and are natural constructs to start when creating a fault model for the language. The fault model identifies kinds of faults we believe are likely to be present in AspectJ programs. The fault types were identified from an analysis of the syntax and semantics of the pointcut and advice constructs, observations of common problems and mistakes among practitioners in the AspectJ community (i.e., in a mailing list, tutorials, and textbooks), and issues pointed out in the research literature.

The model also includes a fault/failure analysis, which states conditions for a fault to execute, for the execution of the fault to cause an infection, and for the infection to propagate to observable output and thereby cause a failure. Pointcut faults turn out to share most properties of infection and propagation, and can be treated uniformly. Three errors resulting from pointcut faults were identified, and each fault type was described in terms of how it appears syntactically in code and which of the three errors it can result in. Advice faults require a less uniform treatment, and individual infection and propagation conditions were provided for each type/category of fault as appropriate.

We believe that the fault model presented is a good foundation for fault seeding, mutation testing, program inspection, and evaluation of testing strategies for AspectJ programs, and we gave preliminary examples to demonstrate the model's suitability for these purposes. We also believe it can be used to devise test adequacy criteria and new testing strategies (other than mutation testing).

This thesis provides only a starting point in the area of testing aspect-oriented programs, and Chapter 6 explored some future research directions. In the immediate future, we believe the following steps should be taken in order to validate and develop the fault model further:

- **Include more language features.** Advice precedence, inter-type declarations, and *if* pointcuts with side-effects should be analyzed for fault types to be included in the fault model.

- **Empirical evaluation.** Experiments should be run to evaluate to what extent the proposed fault model reflects real faults inherent in AspectJ programs. At the very least, case studies should be carried out. The output from the studies should be fed back into the fault model in terms of new and modified fault types and/or categories. Coming up with a good fault model will likely be a iterative process of refinement and evaluation.

- **Devise test adequacy criteria and testing strategies.** Once some confidence in the fault model has been established from empirical studies, it should be used to derive test adequacy criteria. Once a set of criteria has been developed, existing testing strategies should be evaluated against the criteria, and promising strategies developed further. If existing strategies do not perform well, new testing strategies might be considered.

BIBLIOGRAPHY

[1] abc: The aspectbench compiler for aspectj.
URL: http://abc.comlab.ox.ac.uk/introduction. Last checked: 6/29/2006.

[2] The aspectj 5 development kit developer's notebook.
URL: http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html. Last checked: 6/23/2006.

[3] Aspectj 5 quick reference.
URL: http://www.eclipse.org/aspectj/doc/released/quick5.pdf. Last checked: 6/23/2006.

[4] The aspectj programming guide.
URL: http://www.eclipse.org/aspectj/doc/released/progguide/. Last checked: 6/23/2006.

[5] The aspectj project.
URL: http://www.eclipse/org/aspectj/. Last checked: 6/23/2006.

[6] Wikipedia, the free encyclopedia: Article on aspect-oriented programming.
URL: http://en.wikipedia.org/wiki/Aspect_Oriented_Programming. Last checked: 6/23/2006.

[7] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition filters approach. In *ECOOP '92 European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615, pages 372–395. Springer-Verlag, 1992.

[8] R. T. Alexander and J. M. Bieman. Challenges of aspect-oriented technology. In *Workshop on Sofware Quality, International Conference on Software Engineering, Orlando, Florida*, May 2002.

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

[10] R. T. Alexander, J. Offutt, and J. M. Bieman. Syntactic fault patterns in oo programs. In *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems, Greenbelt, Maryland*, pages 193–202. IEEE, December 2002.

[11] P. Anbalagan and T. Xie. Apte: Automated pointcut testing for aspectj programs. In *2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine*, pages 27–32. ACM, July 2006.

[12] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[13] O. Barzilay, Y. A. Feldman, S. Tyszberowicz, and A. Yehudai. Call and execution semantics in aspectj. In *Proceedings of the Foundations on Aspect-Oriented Languages Workshop,*

International Conference on Aspect-Oriented Software Development, Lancaster, UK, pages 19–23, March 2004.

[14] R. V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6:125–252, 1996.

[15] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[16] J. Brichau and M. Haupt, editors. *Survey of Aspect-Oriented Languages and Execution Models*. AOSD-Europe, 2005.

[17] M. Ceccato, P. Tonella, and F. Ricca. Is aop code easier to test than oop code? In *the 1st Workshop on Testing Aspect-Oriented Programs, 4th International Conference on Aspect-Oriented Software Development, Chicago, Illinois*, March 2005.

[18] A. Colyer, A. Clement, G. Harley, and M. Webster. *eclipse AspectJ*. Addison-Wesley, 2005.

[19] R. A. DeMillo, D. Guindi, W. M. McCracken, A. J. Offutt, and K. N. King. An extended overview of the mothra software testing environment. In *Proceedings of the SIGSOFT Symposium Software Testing, Analysis and Verification*, pages 142–151. ACM, July 1988.

[20] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[21] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[22] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[23] M. A. Friedman and J. M. Voas. *Software Assessment: Reliability, Safety, Testability*. Wiley-Interscience, 1995.

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification, Third Edition*. Addison-Wesley, 2005.

[25] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications, Washington, D.C.*, pages 411–428, September–October 1993.

[26] IEEE. Ieee standard glossary of software engineering terminology, September 1990.

[27] W. Isberg. Re: [aspectj-users] if pointcut and side effects. Post to the aspectj-users mailing list. URL: http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg06243.html.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland*, pages 220–242. Springer-Verlag, June 1997.

[29] C. Koppen and M. Störzer. Pcdiff: Attacking the fragile pointcut problem. In *The European Interactive Workshop on Aspects in Software, Berlin, Germany*, September 2004.

[30] R. Laddad. *AspectJ in Action*. Manning Publications Co., 2003.

[31] O. A. L. Lemos and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In *2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine*, pages 33–38. ACM, July 2006.

[32] O. A. L. Lemos, J. C. Maldonado, and P. C. Masiero. Structural unit testing of aspectj programs. In *the 1st Workshop on Testing Aspect-Oriented Programs, 4th International Conference on Aspect-Oriented Software Development, Chicago, Illinois*, March 2005.

[33] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[34] C. V. Lopes. Aspect-oriented programming: An historical perspective (what's in a name?). Technical Report UCI-ISR-02-5, Institute for Software Research, University of California, Irvine, December 2002.

[35] C. V. Lopes and K. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In *Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy*, volume 821, pages 89–99, 1994.

[36] N. McEachen and R. T. Alexander. Distributing classes with woven concerns – an exploration of potential fault scenarios. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, Chicago, Illinois*, pages 192–200, March 2005.

[37] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[38] L. J. Morell. *A Theory of Error-Based Testing*. PhD thesis, University of Maryland, April 1984.

[39] L. J. Morell and R. G. Hamlet. Error propagation and elimination in computer programs. Technical Report 1065, University of Maryland, 1981.

[40] M. Mortensen and R. T. Alexander. An approach for adequate testing of aspectj programs. In *the 1st Workshop on Testing Aspect Oriented Programs, 4th International Conference on Aspect-Oriented Software Development, Chicago, Illinois*, March 2005.

[41] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[42] S. A. A. Naqvi, S. Ali, and M. U. Khan. An evaluation of aspect oriented testing techniques. In *Proceedings of the 1st IEEE International Conference on Emerging Technologies, Islamabad, Pakistan*, pages 461–466, September 2005.

[43] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Robinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering, Hong Kong, PRC*. IEEE, November 2001.

[44] D. J. Richardson and M. C. Thompson. The relay model of error detection and its application. In *Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis*, pages 223–230. IEEE, July 1988.

[45] I. Sommerville. *Software Engineering, 6th Edition*. Addison-Wesley, 2001.

[46] M. Störzer. Analysis of aspectj programs. In *Proceedings of the 3rd German Workshop on Aspect-Oriented Software Development, Essen, Germany*, pages 39–44, March 2003.

[47] M. Störzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary*, pages 653–656. IEEE, September 2005.

[48] M. Störzer and J. Krinke. Interference analysis for aspectj programs. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop, International Conference on Aspect-Oriented Software Development, Boston, Massachusetts*, pages 35–44, March 2003.

[49] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, May 1999.

[50] A. van Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to jhotdraw. Technical Report SEN-R0507, Centrum voor Wiskunde en Informatica, March 2005.

[51] J. Voas, L. Morrel, and K. Miller. Predicting where faults can hide from testing. *IEEE Software*, 8(2):41–48, 1991.

[52] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, Bonn, Germany*, March 2006.

[53] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 180–189. ACM, March 2006.

[54] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspect-oriented programs. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, Taiwan*, pages 560–565, July 2005.

[55] G. Xu. A regression tests selection technique for aspect-oriented programs. In *2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine*, pages 15–20. ACM, July 2006.

[56] G. Xu, Z. Yang, H. Huang, Q. Chen, L. Chen, and F. Xu. Jaout: Automated generation of aspect-oriented unit tests. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference, Busan, Korea*, pages 374–381, November–December 2004.

[57] W. Xu and D. Xu. A model-based approach to test generation for aspect-oriented programs. In *the 1st Workshop on Testing Aspect Oriented Programs, 4th International Conference on Aspect-Oriented Software Development*, March 2005.

[58] W. Xu and D. Xu. State-based testing of integration aspects. In *Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine*, pages 7–13, July 2006.

[59] W. Xu, D. Xu, V. Goel, and K. Nygard. Aspect flow graph for testing aspect-oriented programs. URL: www.cs.ndsu.nodak.edu/˜wxu/research/436-111ljd.pdf. Last Checked: 2006-04-27.

[60] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the 27th International Computer Software and Applications Conference, Dallas, Texas*, pages 188–197, November 2003.

[61] J. Zhao, T. Xie, and N. Li. Towards regression test selection for aspectj programs. In *2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine*, pages 21–26. ACM, July 2006.

[62] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.