2010

Reverse Engineering Aspects to Derive Application Class Models

Irenee Morcos George Magdalla
Cleveland State University

REVERSE ENGINEERING ASPECTS TO DERIVE APPLICATION CLASS MODELS

IRENEE MORCOS GEORGE MAGDALLA

Bachelor of Science in Information Systems

Cairo University, Egypt

May 2005

submitted in partial fulfillment of the requirements for the degree

MASTERS OF SCIENCE IN SOFTWARE ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

May 2010

This thesis has been approved for the

Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Nigamanth Sridhar

_____

Department/Date

_____

Dr. Yongjian Fu

_____

Department/Date

_____

Dr. Wenbing Zhao

_____

Department/Date

To my beloved parents

ACKNOWLEDGMENTS

I would like to thank Dr. Sridhar, my advisor, for all his help and support, throughout the course of this research.

REVERSE ENGINEERING ASPECTS TO DERIVE APPLICATION CLASS MODELS

IRENEE MORCOS GEORGE MAGDALLA

ABSTRACT

Aspects provide a nice way to modify behavior and implement cross-cutting concerns in object-oriented systems. As such, aspects do not have an existence of their own; the application classes that the aspects refer to must be present in order to instantiate the aspects. In this research, we present a reverse engineering approach to generate a minimal class model that has all the structural elements necessary in order to complete exercise a set of aspects.

TABLE OF CONTENTS

CHAPTER I

Introduction

1.1 Introduction

Cross-cutting concerns represent a class of design issues that cannot be completely and properly modularized using traditional object-oriented design methods. Over the last several years, aspect-oriented programming has been used as an effective way of modularizing and implementing such cross-cutting concerns. An aspect can identify one or more structural join points from among a set of classes, and can define additional behavior to be attached at these join points. The join points are identified using pointcuts in the aspect, and the behavior is defined as advice. The aspect(s) are then woven in with the application classes to generate a new set of classes, which are then supplied to the compiler. This modified set of classes includes behavior defined in both the original classes as well as that defined in the aspects.

One important issue to consider with such a setting is to have a way of explicitly defining and identifying the behavioral portions of the woven executable that came from the original class model separately from those portions that came from the

aspects. A related issue is to consider whether or not it is safe to perform a weaving in the first place. Others have looked at these issues [13]. All of this work is still focused on defining specifications for the aspects separately, but aspects cannot be executed separately, and so the only way to check for safety is after the weaving is complete.

In this thesis, we take a different view: Aspect behavior must be identified and verified to be correct before being woven in along with application classes. To support this view, we present a reverse engineering approach to examine a set of aspects, and use this set of aspects to derive a minimal class model that is a structural subset of the class model of the target application. This minimal application that is created is devoid of any real behavior; the classes that are generated will only have the structural elements that the aspects expect to see.

Metaphorically speaking, if the behavior being introduced by a set of aspects is seen as ornaments on a tree, and the pointcuts represent the hooks, the minimal class model that we will create will represent the smallest tree that can appropriately accommodate all the ornaments.

The aspect(s) can be safely woven with this minimal set of empty classes; no safety requirements can be violated in this set of classes, since they do not represent any behavior in the first place. Any behavior exhibited by this new executable is exclusively attributable to the aspect.

We see a number of applications for such a class model harness to exercise a set of aspects. Some of them are test case generation, design pattern mining, and design model validation. We talk more about them in Chapter VII, in the future work section.

1.2 The Problem

There is a need to be able to identify the behavior of the aspect itself. It would really help the developer if he can see how the aspect he is writing will affect the application. Unfortunately, aspects cannot be executed separately from the application. There is no way to test the aspect code by itself, away from the application. Currently, in order to test the aspect code, the aspects are first weaved with the application, so the behavior we are able to test is really the combination of both the behavior of the application combined with the new behavior added by the aspect code. We need a way to examine the aspect behavior by itself independent from the application.

The motivation for this research can be summarized in:

1. To develop a way to identify the aspect behavior by itself, away from the application.

2. To be able to test the aspect code before weaving into the application.

1.3 The Thesis

We defend the following thesis:

Reverse engineering aspects to derive application class models allows us to examine the behavior of the aspects independently from the application code.

1.4 Solution Approach

In order to identify and verify the aspect behavior to be correct before being woven in along with the application classes, we present a reverse engineering approach

to examine the set of aspects written by the developer. We use this set to derive a minimal class model. The derived class model is a structural subset of the class model of the target application. This minimal application that is created does not have any real behavior. The classes that are generated will only have the structural elements that the aspects expect to see, but they are basically empty classes.

The aspect(s) can be safely woven with this minimal set of empty classes; no safety requirements can be violated in this set of classes, since they do not represent any behavior in the first place. Any behavior exhibited by this new executable is exclusively attributable to the aspect.

1.5 Contributions

We make the following contributions in this thesis:

1. We present a way to examine the aspect code by itself, separately from the application.

2. We present a novel reverse engineering approach for deriving class models from a set of aspects.

3. We describe the formal translation rules for such reverse engineering, and briefly describe how these rules can be mechanically and automatically applied.

1.6 Organization of the Thesis

The thesis is organized as follows. In Chapter 2, we present an introduction to aspect-oriented programming. In Chapter 3, we outline the strategies for reverse engineering harness classes from aspects. In Chapter 4, we describe the list of translation rules that are used in deriving the harness class model. In Chapter 5, we go

through a step-by-step example for applying our approach. In Chapter 6, we present salient work from the literature that is related to our own work. In Chapter 7, we conclude and present future work and applications of our approach.

CHAPTER II

Aspect-Oriented Programming

Cross-cutting concerns represent a class of design issues that cannot be completely and properly modularized using traditional object-oriented design methods. A cross-cutting concern is a design issue that spans many classes, such as logging and authentication. Handling these types of issues using traditional object-oriented approaches usually leads to code tangling, scattering, and duplication [15].

Over the last several years, aspect-oriented programming has been used as an effective way of modularizing and implementing such cross-cutting concerns. An aspect can identify one or more structural join points from among a set of classes, and can define additional behavior to be attached at these join points. The join points are identified using pointcuts in the aspect, and the behavior is defined as advice. The aspect(s) are then woven in with the application classes to generate a new set of classes, which are then supplied to the compiler. This modified set of classes includes behavior defined in both the original classes as well as that defined in the aspects.

Applications of aspect-oriented programming are numerous; to name a few: monitoring techniques such as logging, tracing, and profiling, policy enforcement using

system-wide contracts, optimizations such as pooling and caching, design patterns and idioms, implementing thread safety, authentication and authorization, transaction management, and implementing business rules [15].

Early AOP efforts have been done in universities around the globe. Gregor Kiczales, and his team at Palo Alto Research Center (PARC), of Xerox Corporation, were among the early contributors to the AOP. In the late 1990s, they created AspectJ, one of the first AOP implementations. In 1996, Kiczales came up with the term *aspect-oriented programming* [11]. Currently, AspectJ is supported by eclipse.org, an open source community that continues to enhance, and maintain the project.

AOP is implemented using several programming languages, to name a few: Aspect C as an extension of C++, AspectJ for Java. Also, other implementations of AOP exist as libraries PostSharp for C#, and Managed Extensibility Framework for .Net applications. We are using AspectJ implementation of AOP in this thesis.

The following are the most used terms of AspectJ AOP that we will be using throughout the document:

Aspect: "OOP class like": the encapsulating unit of the AOP code.

Join Point: Point of interest where the new behavior is applied.

Pointcut: Defines the set of join points of interest.

Advice: "OOP method like": The new behavior that will be applied at the defined join point(s). The advice can be applied before, after, or around the join point. `before` advice allows us to add behavior before the join point is executed. `after` advice allows us to add behavior after the join point is executed. `around` Advice allows us to modify, or bypass the behavior of the join point.

It is the aspect weaver's responsibility to inject the new behavior described by the advice, at the join points of interest, producing the woven code, which is then

passed to the compiler.

Let us consider logging as an example. Logging each method that is being called in an application is a common cross-cutting concern. But as simple as this issue seems to be, addressing it using OOP means that we need to change the source code of each method that we want to log, to add a code instruction to print the method's signature. This literarily means touching each method of each class in the application.

The following example demonstrates how effectively AOP addresses the logging cross-cutting concern, as opposed to object oriented programming.

```
1 public aspect LoggingAspect {
2   pointcut log():
3     call(* *.*(..))&& !within(LoggingAspect);
4
5   before(): log(){
6     System.out.println("Log:" + thisJoinPoint);
7   }
8 }
```

AspectJ effectively use wild cards to specify join points. In the above example, wild cards are used in a method signature. We use three asterisks: the first one, for all return types, the second means in all classes, and the third means any method name. We also use double dots in the arguments parentheses, which means that the function can take zero or more arguments. So, the pointcut means that we are interested in any method called, outside of the `LoggingAspect`.

This simple example gives us an idea of how powerful AOP is, and demonstrates the urgency of having a way to help the programmer see how his AOP code will affect the application it is written for.

CHAPTER III

Design Overview

3.1 Introduction

In this chapter we discuss our approach in reverse engineering the aspect code into the constructed application code.Our reverse engineering approach allows us to identify the aspect behavior by itself, away from the target application. We start with the aspect code provided by the developer, and we go through it in several passes, to identify the application classes that the aspect expects to see. Once the application classes are identified, we flesh out the application, with the needed code to trigger the aspect behavior. Then we weave the application with the aspect code provided, so we can see the aspect behavior. Our constructed application is a subset of the original application code, but it is devoid from any behavior, guaranteeing that the behavior of the woven application is coming solely from the aspect written by the developer.

3.2 Design Methodology

The key motivation for our work is to be able to consider a set of aspects, and to be able to independently exercise these aspects (separately from the target application). Based on this, our process of reverse engineering aspects includes the following steps:

1. We identify the list of necessary join points to be included in the class model. We scan through the aspect code in several passes until there are no ambiguities remaining to be revealed. Based on this scan, we identify the structural elements, i.e. packages, classes, data members, constructors, and methods. In addition, we discover the relationship between these structural elements, if any.

2. We flesh out the application by creating the packages, classes, data members, methods and constructors, such that all structural relationships are maintained.

3. We create a trigger method in each created class to call all the methods that were defined for that class, to exercise the aspect, and trigger the aspect behavior.

4. We create a separate "Helper" class, with a main function to call the invoking methods of the created classes. We make sure the trigger methods and the "Helper" class we added, are transparent to the aspect, since they are not part of the original application. So we rewrite the pointcuts to exclude them.

The following is an example of the reverse engineering process:

We first start with the aspect that is provided by the developer.

```
1 public aspect AccountAspect {
2   pointcut creditOperation(Account account, float amount):
3     call( void Account.credit(float)) &&
4       this(account) && args(amount);
5
6   before( Account account, float amount):
7     creditOperation(account, amount) {
8     System.out.println(
```

```
9         "Crediting␣" + amount +"␣to␣" + account );
10    }
11
12    void around(Account account, float amount) throws
13        InsufficientBalanceException:
14      call( * Account.debit(float) throws
15          InsufficientBalanceException)
16          && target(account) && args(amount)
17    {
18      try {
19        proceed(account, amount);
20      } catch(InsufficientBalanceException ex) {
21        System.out.println("Overdraft␣protection␣logic");
22      }
23    }
24
25    after() returning : call(* Account.*(..)){
26      System.out.println(
27        "Log␣the␣successful␣completion" + thisJoinPoint);
28    }
29  }
```

This aspect includes, the three different types of advices: `before`, `around`, and `after`. It also includes one named pointcut `creditOperation(Account account, float amount)`, and two anonymous pointcuts.

The first pointcut has two arguments, one of type `Account`, and another of type `float`. We check if `Account` is a system-defined class, or if it was defined before. Since `Account` is not a system-defined class, and has not been defined before, we add it to the list of classes to be created. Since `float` is a system-defined type, we do not add it to the list. The pointcut specifies the join point where it will be applied: on the call of the method of which the signature is provided: `void Account.credit(float)`. This means that the `Account` class that we are creating has a method called `credit()`, that takes an argument of type `float`, and has a `void` return type. So, we add this method to the methods that will be created for the `Account` class. The `this` and `args` keywords are used to pass context to the advice, that uses this pointcut.

Lines 6 to 10 is an advice, it is a `before` advice, which means that the behavior defined in this advice will be executed before the join points to where the advice is being applied. It has two arguments; those are of the pointcut it is applied to. The

context is passed using the arguments, so the advice will print how much is being credited to which account.

Lines 12 to 23 is an `around` advice. The pointcut for this advice is anonymous. The join points are defined in the advice itself. `Account` is already in the list of classes to be created, so we do not need to add it. The join points are also defined using a call pointcut of the method whose signature is `* Account.debit(float) throws InsufficientBalanceException)`. `InsufficientBalanceException` is added to classes to be defined, it is flagged as an exception, so it will be declared to extend `Exception`. Since the join point throws an exception, the advice is declared to throw the exception also. `proceed` is a very important keyword of the `around` advice; without it the join point will not get executed. Unlike the `before` and `after` advice, the `around` advice has a return type, and it is the same as that of `proceed`. This means that the `Account` class has a method called `debit()` that takes one argument of type `float`, has `void` as return type, and throws an `InsufficientBalanceException`. As with regular Java, since `debit()` throws an exception, `proceed` is called between `try` and `catch`.

Lines 25 to 28 is an `after` advice. `returning` specifies that this advice only applies to successfully executed join points. So, it will not be executed if a join point throws an exception. This advice uses an anonymous pointcut as well. The join points are defined using wild cards. The method signature has 3 wild cards, the first `*` for any return type, second `*` for any method in the `Account` class, and the third `..` for zero or more arguments. This means that the advice will be applied on any method in `Account` that is being called. Since we have at least one method defined for class `Account`, we do not need to define new methods. A call to any method of `Account` class will satisfy/exercise this advice. `ThisJoinPoint` is a keyword that is used to print the current join point signature.

Now we have the list of classes and methods to be defined. So we can flesh out the application. We create class Account and its two methods `debit()` and `credit()`.

```
1 public class Account {
2   void debit(float f) throws InsufficientBalanceException{
3     if(Math.random() > 0.5)
4       throw new InsufficientBalanceException();
5   }
6
7   void credit(float f){}
8 }
```

Since the `debit()` method throws an exception we use a random number to trigger the exception.

We create InsufficientBalanceException as follows:

```
1 public class InsufficientBalanceException extends Exception{
2 }
```

In order to see the aspect behavior, we need to call all the methods that we have defined. To do so we need a method that does exactly that: we need it inside of the created class, in case we have private members that need to be called.

```
1 public class Account {
2   void debit(float f) throws InsufficientBalanceException{
3     if(Math.random() > 0.5)
4       throw new InsufficientBalanceException();
5   }
6
7   void credit(float f){}
8
9   public void trigger(){
10    Account i1 = new Account();
11    i1.credit(0);
12    try {
13            i1.debit(0);
14    } catch(Exception e) {}
15  }
16 }
```

We call the methods of the class using arguments' default values. `InsufficientBalanceException` does not need an `trigger()` since it does not have any methods. It is the responsibility of the `Helper` class to call all the `trigger`s defined.

```
1 public class Helper {
```

```
2   public static void main(String[] args) {
3     Account a = new Account();
4     a.trigger();
5   }
6 }
```

The output is:

```
Crediting 0.0 to Account@1389e4
Log the successful completioncall(void Account.credit(float))
Overdraft protection logic
Log the successful completioncall(void Account.trigger())
```

As we can see the first line is from the `before` advice, the second and fourth are from the `after` advice, and the third line of output is from the `around` advice. From the output we can see that calling the `debit()` function threw the exception, which was caught displaying `Overdraft protection logic`. Another thing to notice is that the advice was also applied to the `trigger()`.

Since the `after` advice applies to all called methods of the `Account` class, we need to exclude our added `trigger()`. We only added this method to trigger the aspect behavior, but it should be transparent to the developer, since it is not part of the application. This is also true for class `Helper`, so we need to not apply the aspect on our `Helper` class. To do so, we modify the aspect provided by the developer and define 2 pointcuts to exclude them as follows:

```
1 public aspect AccountAspect {
2   pointcut excludeMyHelperClass():
3     !within(Helper);
4
5   pointcut excludeMyMethodsFromCall():
6     !call (void *.trigger() );
7
8   pointcut creditOperation(Account account , float amount):
9     call( void Account.credit(float)) && this(account) && args(amount);
10
11  before( Account account , float amount):
12      creditOperation(account , amount) && excludeMyHelperClass(){
13    System.out.println("Crediting␣" + amount +"␣to␣" + account );
14  }
```

```
15
16   void around(Account account , float amount)
17       throws InsufficientBalanceException:
18     call (* Account.debit(float) throws InsufficientBalanceException)
19         && target(account) && args(amount) && excludeMyHelperClass(){
20       try {
21         proceed(account , amount);
22       } catch(InsufficientBalanceException ex){
23       System.out.println("Overdraft␣protection␣logic");
24   }
25
26   after() returning :
27     call(* Account .*(..)) && excludeMyHelperClass() &&
28         excludeMyMethodsFromCall() {
29       System.out.println("Log␣the␣successful␣completion"
30         + thisJoinPoint);
31   }
32 }
```

After this modification the output is:

```
Crediting 0.0 to Account@c20e24

Log the successful completioncall(void Account.credit(float))

Log the successful completioncall(void Account.debit(float))
```

We can see that this time, the after advice is not applied to our `trigger()`. Another thing to note is that the `debit()` method did return successfully.

CHAPTER IV

Transformation Rules

4.1 Introduction

In this chapter we examine the transformation rules we set to reverse engineer the aspects into application classes. We developed our transformation rules for AspectJ language [1]. AspectJ5 is the most current version of AspectJ. We believe that our approach will stand applicable for future versions of AspectJ, since we have based our approach on searching for the elements of the target application that are mentioned in the aspect code, and the aspect needs to specify the join points to where the behavior gets attached. We start with the aspect source code, and we use the transformation rules listed in this chapter to construct the minimal set of classes that is needed to exercise the aspects. These rules are set such that the constructed application classes have no behavior in themselves; the only behavior in the woven application is solely coming from the aspects, provided by the aspect developer.

We categorize the rule by the object-oriented element it defines, so we have rules for defining packages, classes, constructors, methods, and fields. We show how

we use the clues found in the aspect source code to construct the minimal set of application classes.

We present how each clue is transformed into the constructed object oriented code. An important thing to note is that we do not define any Java system-defined classes such as `void`, `int`, `float`, `double`, `long`, `Object`, `Integer`, `Float`, `Double`, `Long`, `String`, `Math` library, etc. We assume that the developer imports the appropriate classes to use system-defined classes, for example `java.util` to use `List`, `ArrayList`, `Map`, etc.

We also use the Java Reflection API [20] to determine if a class is system-defined, and to get information about that class. Reflection API is a library that allows us to examine the classes at runtime in Java Virtual Machine.

Also, we ignore any intertype declarations, or hierarchies introduced by the aspect itself. If the aspect is introducing a new field or a method to a class of the application, we do not add this element to the class, since this introduction is done in the aspect and not part of the original application.

We illustrate the rules with examples on how we spot the clues in the aspect source code, and what is the output of this clue in the constructed application classes code. For organization, we name the rules used to derive packages as Pkg#, Cls# for classes, Mtd# for methods, Ctr# for constructors, and Fld# for fields.

4.2 Packages

Packages can be spotted in the aspect code in the `import` statements, or in the ClassName, FieldSignature, MethodSignature, or ConstructorSignature as part of their fully qualified name. We assume the following format for a fully qualified name:

```
1 package1.package2...packageN.ClassName.FieldName/MethodName/new
```

The following are the rules we use to create packages in the constructed application code.

Pkg1 – `PackageName.ClassName`

Examples:

```
within(userPackage.Class1)
import(java..*)
within(userPackage..*)
```

This clue tells us that `PackageName` is a package in the application. We check if the package is a system-defined package, e.g. `java`, and if so, we do not add it to the list of packages to be created. We assume that the appropriate system-defined classes are imported with the aspect code. If `PackageName`, e.g. `userPackage`, has not been added to the list of packages to be created, we add it. This clue is used to define package `userPackage` as follows:

```
package userPackage;
```

Pkg2 – `PackageName.PackageName.ClassName`

Example:

```
import package1.package2.*
```

This statement imports all classes in `package2`, and `package2` is a package inside `package1`. If not already defined, we define `package1.package2`. This statement is transformed as follows:

```
package package1.package2;
```

Pkg3 – `userPackage*..*`

Example:

```
within(userPackage*..*)
```

This pointcut gives us the clue that there are one or more packages which names start with `userPackage`. If we did not add `userPackage`, or any other package that starts with `userPackage`, to the list of packages to be defined, we need to do that. The constructed application will have the following line of code, built on this clue, satisfying the pointcut.

```
package userPackage;
```

Pkg4 – `PackageName.*.ClassName`

Example:

```
within(userPackage.*.Class1)
```

This pattern matches all classes with name `Class1`, that exist in a subpackage of `userPackage`. We add `userPackage` to the list of packages to be created, if it is not already defined. If it is already added, we check if it has any sub package already created. If we do not find any sub package for `userPackage`, we create a `package_p#` inside `userPackage`. The following is how these clues are transformed in the constructed application classes:

```
package userPackage.package_p1;
```

4.3 Classes

There are several pointcuts that are used to define classes in the application. We can also spot classes in the fully qualified name of a field, method, or a constructor. The following are the statement and pointcuts where we spot the classes necessary to construct the application code.

```
import PackageName.ClassName;
```

This statement includes the mentioned classe(s) to the source code.

```
staticinitialization(ClassName)
```

This pointcut matches the static initialization of `ClassName`. When `ClassName` is initialized after loading.

```
declare parents: ClassName1 extends ClassName2;
```

This pointcut tells us that is the woven code that `ClassName1` is a child of `ClassName2`.

```
within(ClassName)
```

This pointcut is usually used with another pointcut. It states that the other pointcut should match within the code of the mentioned class.

```
this(ClassName/Class-Instance)
```

This pointcut is usually used with another pointcut. It states that the other pointcut should match where the object is of type `ClassName`.

```
target(ClassName/Class-Instance)
```

This pointcut is usually used with another pointcut. It states that the other pointcut should match where the target object is an instance of `ClassName`.

```
args(ClassName/Class-Instance)
```

This pointcut is usually used with another pointcut. It states that the other pointcut should match where the arguments are of type `ClassName`.

```
after() returning(ClassName Class-Instance)
```

This statement could be part of an after advice. It states that the pointcut where the advice is applied, returns an object of type `ClassName`.

```
handler(ExceptionType)
```

This pointcut gets exercised when the `ExceptionType` is handled.

```
after() throwing(ExceptionType Exception-Instance)
```

This statement could be part of an after advice. It states that the pointcut where the advice is applied, throws an exception of type `ExceptionType`.

```
throws(ExceptionType)
```

This statement may be a part advice, method or constructor signature, it states that an exception of type `ExceptionType` may be thrown.

```
ClassName
```

Could be only the class name or the fully qualified name of the class, i.e. includes the package where the class is declared; in which case we would follow the package rules described earlier to identify the package, and create the class inside the appropriate package. Note that if there is no package name associated with the class name, we create the class in the default package where the aspect code resides.

Following are the rules we use to spot classes in the aspect code, and how we use them in the constructed application code. The same as for packages, if the class is system-defined, we do not need to create it; and if it is not we need to create it in the constructed application code.

Cls1 − `ClassName`

Example:

```
staticinitialization(Class1)
after() returning(Class1 c)
```

The static initialization pointcut matches `Class1`, and the `returning` clause states that there is a `Class1` class in the application. We first check if `Class1` is a

Java-defined class, if not we check if we have added it to the list of classes to be defined, if we did not, then we add it. This will result in the following code to be created, as part of the application classes.

```
class Class1{
...
}
```

Cls2 – `ClassName || ClassName`

Example:

```
within(Class1 || Class2)
```

This clue tells us that the application has classes with names: `Class1`, and `Class2`. If any of them is not in the list of classes to be created, we add it. This clue will be transformed as follows in the constructed application classes:

```
class Class1{
...
}
class Class2{
...
}
```

Cls3 – `ClassName*`

Example:

```
within(Class1*)
```

This clue says that there are classes whose names start with `Class1`. Having a class named `Class1` in the constructed application satisfies this pointcut. So, we check if we have a class called `Class1`, or a class whose name starts with `Class1`, in the list of classes to be created, if we do not find one, we add Class1 to the list. The following should be part of the constructed application code:

```
class Class1{
```

```
2 ...
3 }
```

Cls4 – `*ClassName`

Example:

```
1 within(*Class1)
```

This statement says within classes which names end in `Class1`, so having a class named `Class1` in the constructed application satisfies this pointcut. So, we check if we have not added a class `Class1`, or another class whose name ends with `Class1`, to the list of classes to be created we add `Class1` to it. The following should be part of the constructed application code:

```
1 class Class1{
2 ...
3 }
```

Cls5 – `PackageName.ClassName`

Example:

```
1 within(userPackage.Class1)
```

This statement says that there is a `Class1` class in `userPackage` package. If `userPackage` is not in the packages list to be defined, add it, as described in Section 4.2. And add `Class1` to the list of classes to be created inside `userPackage`. These clues are transformed into the following application code:

```
1 package userPackage;
2 class Class1{
3 ...
4 }
```

Cls6 – `ClassName+`

Example:

```
within (Class1+)
```

This clue tells us that a pointcut will be attached to `Class1` and its children classes. Having `Class1` satisfies this pointcut. We add `Class1` to the list of classes to be created, if it is not have been added before. The following is how this clue is translated in the constructed application code:

```
class Class1{
...
}
```

Cls7 – `this(ClassName/class instance), target(ClassName/class instance), args(ClassName/class instance)`

Example:

```
this(Class1)
   target(Class1)
   args(Class1)
```

Or

```
Class1 c;
this(c)
target(c)
args(c)
```

These pointcuts match class `Class1`. We add `Class1` to the list of classes to be created, if it is not already there. This clue will be transformed into the following in the constructed application classes:

```
class Class1{
...
}
```

Cls8 – `handler(ExceptionType)`

Example:

```
handler(MyException+)
    handler(*MyException)
    hander(MyException*)
    handler(MyException)
```

The first pointcut matches handling `MyException` and all its subclasses. The second pointcut matches handling all exceptions that end with `MyException`. The third pointcut matches handling all exceptions that start with `MyException`. The fourth pointcut matches handling `MyException`. Satisfying the fourth pointcut satisfies all the other. This clue is used as follows in the constructed application code:

```
class MyException extends Exception{
...
}
```

Cls9 – `throws(ExceptionType)`

Example:

```
throws(MyException)
```

This clue states that there is an exception of type `MyException`. we basically treat exceptions as classes that are flagged to be exceptions, so we can declare it to extend the Exception class when we flesh out the application. If `MyException` is not added to the list of exceptions to be created, add it. We basically treat exceptions as classes that are flagged to be exceptions, so we can declare it to extend the Exception class when we flesh out the application. The following is how this clue is transformed in the constructed application code:

```
class MyException extends Exception{
...
}
```

Cls10 – `after() throwing(ExceptionType exceptionObject)`

Example:

```
after()throwing(MyException)
```

This clue states that there is an exception of type `MyException`. If `MyException` is not added to the list of exceptions to be created, then we add it. The following is how this clue is transformed in the constructed application code:

```
class MyException extends Exception{
...
}
```

Cls11 – `after() returning(ClassName Class-Instance)`

Example:

```
after()returning(Class1 c)
```

This clue states that there is a class of type `Class1`. If `Class1` is not added to the list of classes to be created, add it. The following is how this clue is transformed in the constructed application code:

```
class Class1{
...
}
```

Cls12 – `after() returning`

The return type is not specified, so do not do anything.

Cls13 – `declare parents: ClassName1 extends ClassName2`

Example:

```
declare parents: SubAccount extends Account
```

This is a way of how the aspect can change the hierarchy of the classes of the original application; it can define one to be a subclass of the other. This clue gives us the names of two classes in the application that need to be defined as follows:

```
class SubAccount{
...
}

class Account{
...
}
```

Notice that we do not define any hierarchy to the classes, since the aspect introduces the hierarchy.

Cls14 – `PackageName.*`

Example:

```
within(PackageName.*)
```

This pointcut tells us that the other pointcut associated with this one is applicable to all classes inside `PackageName`. If we have at least one class in the list of classes to be created in the mentioned package, we do not do anything. If no classes are defined `PackageName`, we create a new class called `ClassCls#`. The following is how the clue is transformed in the constructed application code.

```
package PackageName;
class ClassCls1{
...
}
```

Cls15 – `!within(ClassName) , !this(ClassName), !target(ClassName)`

Example:

```
!within(Account)
```

The pointcut matches all classes except the `Account` class. If there is no other class in the list to classes to be created, add `ClassCls#` to it if there is already at least one other class defined do not add anything, the other classes will satisfy this pointcut. This clue may be transformed into:

```
class ClassCls1{
...
}
```

Cls16 – `!ClassName`

Example:

```
within(!Account)
```

The pointcut matches all classes except the `Account` class. Keep as ambiguity for next pass and if there are no classes defined other than Account, add `ClassCls#` to the list of classes to be created, if there is already at least one other class defined do not add anything, the other classes will satisfy this pointcut. This clue may be transformed into:

```
class Class1{
...
}
```

Cls17 – **Inheritance**

If a system-defined class is used, and it is overridden by a new user-defined class, then the user-defined class is a child of system-defined. All standard inheritance rules should apply.

Example:

```
import java.awt.*;
public aspect AccountAspect {
  void aspectMethod()
  {
```

```
5       RunnableWithReturn worker = new RunnableWithReturn();
6       try {
7         EventQueue.invokeAndWait(worker);
8       } catch (Exception ex){}
9     }
10  }
```

`invokeAndWait()` takes an argument of type `Runnable`, and since `Runnable` is an interface then `RunnableWithReturn` must implement it and its abstract methods. The following is how `RunnableWithReturn` should be defined:

```
1  public class RunnableWithReturn implements Runnable{
2    public void run() {
3
4    }
5  }
```

4.4 Fields

The pointcuts that define fields or data members are `set` and `get`. They take the following format:

```
1  set(FieldSignature)
```

The join point defined by this pointcut is the field mentioned in the `FieldSignature`. The associated behavior is seen on its initialization or when it is assigned a value.

```
1  get(FieldSignature)
```

The join point defined by this pointcut is the field mentioned in the `FieldSignature`. The associated behavior is seen accessing this field to get its value, as when returning the field, or using its value to define another variable.

`FieldSignature` has the following format:

```
1  AccessModifier Type ClassName.FieldName
```

Example:

```
1 private float Account.balance;
```

Notice that if the `ClassName` is part of the field signature, and has not been already added to the list of classes to be created, we add it, following the rules discussed in Section 4.3. The same is also done if the package is defined in the method signature. E.g.: `private float banking.Account.balance`. In this case we add `banking` to the list of packages to be created, and add `Account` to the list of classes to be created in the `banking` package.

In order to see the associated behavior added by the aspect, we access the field defined in a method (named `trigger()`) that is created for each created class. We define `trigger()` as a `public` method that takes no arguments and has a `void` return type. The responsibility of this method is to call/access all the class members that have been defined for the created classes, in order to trigger the associated behavior specified by the aspect. When setting the fields, we use the following default values for each type:

- `short, int, long, float, double : 0`
- `boolean : false`
- `String : ''`
- `Object : new Object of the specified type`
- `Array : call with new Object[1] of the specified type.`

When we encounter a `set`, we add the field, with the `set` flag set to true, to the list of fields to be added for the mentioned class, if it has not been already added. We initialize the field with the appropriate default value in the `trigger()` method. When we encounter a `get`, if the field is not already added to the list of fields to be created for the mentioned class, we add it, with the `get` flag set to true. We assign the field to another local variable of the same type as the field in the `trigger()` method.

The following is the list of rules that we use to decipher the field signature clues to construct the application classes.

Fld1 – `AccessModifier Type ClassName.FieldName`

Same with: `set/get(AccessModifier type FieldName) && within(ClassName)`

This clue tells us, that there is `FieldName` with the mentioned access modifier, of type `Type`, a data member of class `ClassName`. The following is how the clue is used in the constructed code application, for `set` and `get`:

Example:

```
1 set(private float banking.Account.balance)
```

This clue is transformed into the following in the constructed application:

```
1 package banking;
2 class Account{
3    private float balance;
4    public void trigger(){
5       balance = 0;
6    }
7 }
```

Example:

```
1 get(private float banking.Account.balance)
```

This clue is transformed into the following in the constructed application:

```
1 package banking;
2 class Account{
3    private float balance;
4    public void trigger(){
5       float i1 = balance;
6    }
7 }
```

Fld2 – `AccessModifier Type ClassName.*`

This clue tells us, that this pointcut applies, to all the fields of `ClassName`. If it is a `set`/`get` pointcut, we look for any field with of the specified type, with the `set` flag

set to `true`. If there is one, it will satisfy the point cut. Otherwise we add the field with the `FieldSignature` to the list of fields to be created for the mentioned `ClassName`, and set its `set`/`get` respectively flag to `true`. The following is how the clue is used for `set`/`get` pointcuts, into the constructed application code:

Example:

```
set(protected int Account.*)
```

This clue is transformed into the following in the constructed application:

```
class Account{
   protected int f1;
   public void trigger()
   {
       f1 = 0;
   }
}
```

Example:

```
get(protected int Account.*)
```

This clue is transformed into the following in the constructed application:

```
class Account{
   protected int f1;
   public void trigger()
   {
       int i1 = f1;
   }
}
```

Fld3 – `AccessModifier * ClassName.*`

This clue tells us that the pointcut is interested in all fields regardless of their names, or types, but these, which have the mentioned `AccessModifier`. If we find a field with the mentioned `AccessModifier`, and the appropriate `set`/`get` flags turned on, we do not do anything. Otherwise, we add a field of type `int`, with the rest of the mentioned specifications, and with `set`/`get` flags set appropriately.

Example:

```
set(private * Account. *)
```

This clue is transformed into the following in the constructed application:

```
class Account{
   private int f1;
   public void trigger()
   {
       f1= 0;
   }
}
```

Example:

```
get(private * Account.*)
```

This clue is transformed into the following in the constructed application:

```
class Account{
   private int f1;
   public void trigger()
   {
       int i1 = f1;
   }
}
```

Fld4 – `!AccessModifier ClassName.FieldName`

The clue tells us that this pointcut does not apply on the specified access modifier. If there is a field that is not with the mentioned access modifier, we add one. The same rule applies for `!final`, `!static`.

4.5 Methods

Pointcuts such as `call`, `execution`, `within`, `cflow`, and `cflowbelow` are used to define method join points. We depend on the signature used in the pointcut, to determine if the join point is a constructor or a regular method. These pointcuts are written in the following format:

```
execution(MethodSignature)
```

The join point defined by this pointcut is the method mentioned in the `MethodSignature`. The associated behavior is seen, when the method is executed at the class side.

```
call(MethodSignature)
```

The join point defined by this pointcut is the method mentioned in the `MethodSignature`. The associated behavior is seen, when the method is called. Unlike in an execution pointcut, the advice behavior is inserted at the caller side, instead of callee side.

```
withincode(MethodSignature)
```

The join point defined by this pointcut resides in the body of the method defined with `MethodSignature`.

```
cflow(pointcut)
```

This pointcut takes another pointcut as an argument. The join points defined by the `cflow` pointcut are all the joint points in the control flow of the enclosed pointcut, including the calling join point.

```
cflowbelow(pointcut)
```

This pointcut takes another pointcut as an argument. The join points defined by the `cflowbelow` are all the join points in the control flow of the enclosed pointcut, excluding the calling join point.

`MethodSignature` has the following format:

```
AccessModifier ReturnType ClassName.MethodName(arg1,..., argN)
    [throws ExceptionType]
```

Example:

```
public float Account.getBalance()
```

Notice that if the `ClassName` is part of the method signature, and has not been already added to the list of classes to be created, we add it, following the rules discussed in Section 4.3. The same is also done, if the package is defined in the method signature. E.g.: `public float banking.Account.getBalance()`. In this case, we add `banking` to the list of packages to be created, and add `Account` to the list of classes to be created in the `banking` package.

If the method signature specifies that the method throws an exception, we add the `ExceptionType` to the list of exceptions to be created, as described in Section 4.3.

In order to see the associated behavior added by the aspect, we call each defined method in a function (named `trigger()`) that is created for each created class. We define `trigger()` as a `public` method that takes no arguments and has a `void` return type. The responsibility of this method is to call/access all the class members that have been defined for the created classes, in order to trigger the associated behavior specified by the aspect. When calling methods or constructors, we use the following default values for each type:

- `short, int, long, float, double : 0`
- `boolean : false`
- `String : ‘‘’’`
- `Object : new Object of the specified type`
- `Array : call with new Object[1] of the specified type.`

When calling a class member that throws an exception, we call it inside a `try` and `catch` block.

The following is the list of rules that we use to decipher the method signature clues to construct the application classes.

Mtd1 – `AccessModifier ReturnType ClassName.MethodName(arg1,..., argN)`

Example:

```
call(public float Account.getBalance(AccountNumber, int))
```

This clue tells us that class `Account`, has a `public` method named `getBalance()`, that takes two arguments, one of type `AccountNumber`, and the other of type `int`, and it has `float` as its return type. We use the rules in Section 4.3 to determine if we need to add `Account` and `AccountNumber` to the list of classes to be created. If the class, `Account` in this example, does not have a method with this signature, we add the method to the list of methods to be created for the `Account` class. The following is how this clue is transformed into the constructed application code:

```
class Account{
   public float getBalance(AccountNumber a1, int a2){
      return 0;
   }
   public void trigger(){
      getBalance(new AccountNumber(), 0);
   }
}
class AccountNumber{
}
```

Mtd2 – `AccessModifier ReturnType ClassName.MethodName(arg1,..)`

- `AccessModifier ReturnType ClassName.MethodName(.., argN)`

- `AccessModifier ReturnType ClassName.MethodName(arg1,.., argN)`

Example:

```
private int Class1.method1(int,..)
private int Class1.method2(.., float)
private int Class1.method3(.., long, ..)
```

These clues are saying that these methods have an argument of type `arg#`, and may have others, that the pointcut does not care about. This means that we can treat these examples, as their following correspondents:

```
private int Class1.method1(int)
private int Class1.method2(float)
private int Class1.method3(long)
```

This means we can use rule Mtd1 to create the appropriate methods, simply by ignoring the ".." part of the arguments.

Mtd3 – `AccessModifier ReturnType ClassName.MethodName(arg1,..., argN) throws ExceptionType`

Example:

```
call(private void Account.debit(float) throws InsufficentBalanceException)
```

Or:

```
after() throwing(InsufficentBalanceException e):
call(private void Account.debit(float){
   System.out.println( After throwing exception);
}
```

This clue tells us that the `Account` class has a private method called `debit()` that takes a `float` as an argument and returns a void type, and throws an `InsufficentBalanceException`. We use class rules to decide whether we need to create `Account` and `InsufficentBalanceException`. If this method's unique signature is not already included, it is added to the list of methods to be created for the enclosing class, `Account` in this case. Each created method is called in the public `trigger()` method, and only the `trigger()` method, is called from outside of the class, in the `Helper` class. The following is how we use the clue in the constructed application code:

```
class Account{
   private void debit(float a1) throws InsufficentBalanceException{
      if(Math.random() > 0.5)
         throw new InsufficentBalanceException();
   }
   public void trigger(){
      try{
         debit(0);
      }catch(Exception e){}
   }
}
```

```
12 class InsufficentBalanceException extends Exception{
13 }
```

Mtd4 – `execution(AccessModifier ReturnType MethodName(arg1,..., argN)) && within(ClassName)`

Example:

```
1 execution(public int method1(AccountNumber)) && within(Account)
```

This clue says that `Account` class has a `public` method, that takes an argument of type `AccountNumber`, and returns an `int` return type. If the `ClassName` class does not have a method with this unique signature, we add one to the list of methods to be created for this `ClassName`. The following is how this clue is used in the constructed application code.

```
1 class Account{
2    public int method1(AccountNumber a1) {}
3   public void trigger(){
4    method1(new AccountNumber());
5   }
6 }
7 class AccountNumber{
8 }
```

Mtd5 – `pointcut(MethodSignature) && within(ClassName || ClassName)`

Example:

```
1 call(private int method1()) && within(Account || SavingsAccount)
```

This clue tells us that `method1()` is either found in `Account` or `SavingsAccount`. We check the list of methods to be created for both methods. If `method1()` is not already added at either, we add it to the first class, `Account` in this case. The following is how this clue is used in the constructed application code.

```
1 class Account{
2   private void method1(){}
```

```
3   public void trigger(){
4     method1();
5   }
6
7 class SavingsAccount{
8 }
```

Mtd6 – `call(MethodSignature1) && withincode(MethodSignature2)`

Example:

```
1 call(public void AccountNumber.method1(int)) &&
2     withincode(private int  Account.method1())
```

This clue tells us that `MethodSignature1` is called from `MethodSignature2`. Using the method signatures we create methods in the appropriate classes. We add a call to `MethodSignature1` in `MethodSignature2`'s body and we make sure we import the appropriate package where `MethodSignature1` resides, if it is different than `MethodSignature2`. We use the appropriate way to create an object of the class where `MethodSignature1` is defined. The following is how this clue is used in the constructed application code:

```
1 class AccountNumber{
2    public void method1(int i){}
3    public void trigger(){
4       method1(0);
5    }
6 }
7 class Account{
8    private int method1(){
9       AccountNumber i1 =  new AccountNumber();
10      i1.method1();
11   }
12   public void trigger(){
13      method1();
14   }
15 }
```

Mtd7 – `call(ConstructorSignature) && withincode(MethodSignature)`

Example:

```
1 call(public Class1.new()) && withincode(private void Class2.method1(int))
```

This clue says that an object is initialized using `ConstructorSignature` in `MethodSignature`. We use the rules in Section 4.3 to create the classes, and the rules in Section 4.6 to create the constructor. We create `MethodSignature` in the appropriate class. We initialize an object using `ConstructorSignature` in `MethodSignature`'s body. We make sure we import the appropriate package, if `ConstructorSignature` and `MethodSignature` are not in the same package. The following is how this clue is transformed in the constructed application code:

```
class Class1{
   public Class1(){}
   public void trigger(){
      Class1 i1 = new Class1();
   }
}
class Class2{
   private void method1(int i1){
     Class1 i1 = new Class1();
   }
   public void trigger() {
       method1(0);
   }
}
```

Mtd8 – `AccessModifier ReturnType ClassName.MethodNamePart*(arg1,..., argN)`

Example:

```
public void Account.new*()
```

This clue tells us that the `Account` class has a public method whose name starts with `MethodNamePart`, `new` in this case, and takes no arguments, and returns a `void` type. We check the list of methods to be created for the `ClassName`, if we do find a method whose name starts with `MethodNamePart`, and has the exact method signature except for other part of the name, we do not do anything, and the found method satisfies this pointcut. Otherwise, we add a new method called `MethodNamePart_m#`, with the rest of the defined specification in the `MethodSignature`. The following is how this example clue is transformed in the constructed application code.

```
class Account{
   public void new_m1(){}
   public void trigger(){
      new_m1();
   }
}
```

Mtd9 – `AccessModifier ReturnType ClassName.*MethodNamePart(arg1,..., argN)`

Example:

```
private void Account.*int()
```

This clue tells us that the `Account` class has a `public` method whose name ends with `MethodNamePart`, `int` in this case, and takes no arguments, and returns a `void` type. We check the list of methods to be created for the `ClassName`, if we do find a method whose name ends with `MethodNamePart`, and has the exact method signature except for other part of the name, we do not do anything, the found method satisfies this pointcut. Otherwise, we add a new method called `m#_MethodNamePart`, with the rest of the defined specification in the `MethodSignature`. The following is how this example clue is transformed in the constructed application code.

```
class Account{
   public void m1_int(){}
   public void trigger(){
      m1_int();
   }
}
```

Mtd10 – `AccessModifier ReturnType`

`ClassName.MethodNamePart1*MethodNamePart2(arg1,..., argN)`

Example:

```
public void add*Listener()
```

This clue tells us that the `Account` class has a `public` method whose name starts with `MethodNamePart1` (`add`) and ends with `MethodNamePart2` (`Listener`), and takes no

arguments, and returns a `void` type. We check the list of methods to be created for the `ClassName`, if we do find a method which name starts with `MethodNamePart1`, and ends with `MethodNamePart2`, and has the exact method signature except for other parts of the name, we do not do anything, the found method satisfies this pointcut. Otherwise, we add a new method called `MethodNamePart1MethodNamePart2`, with the rest of the defined specification in the `MethodSignature`. The following is how this example clue is transformed in the constructed application code.

```
1 class Account{
2    public void addListener(){}
3    public void trigger(){
4       addListener();
5    }
6 }
```

Mtd11 – `AccessModifier * ClassName.MethodName(arg1, ..., argN)`

Example:

```
1 public * Account.getBalance()
```

This clue tells us that the pointcut enclosing `MethodSignature` is satisfied with `ClassName.MethodName` method, that has the mentioned access modifier, and arguments' types and order, returns any return type. There is another clue we need to check for in order to determine the return type. We need to check if this pointcut is mentioned in an `around` advice, we use the return type of the `around` advice as the return type of the method. We also do the same for `after returning` advice, which specifies the return type. We check if the list of methods to be created for the `ClassName` class, and if the rest of the method specifications found in the `MethodSignature` matches any of the methods to be created, we do not do anything. Otherwise we add the method to the list of methods to be created for `ClassName` class, and we use the return type found for the `around` or `after returning` advice as the return type of the method. If the is no `around` advice associated with this pointcut, we make the return type `void`.

```
int around():
   execution(public * Account.getBalance()){
 return proceed();
}
```

Or:

```
after() returning(int i)
   execution(public * Account.getBalance()){
   System.out.println(``After returning advice'');
}
```

If we have an `around` or `after returning` advice, the clue is used as follows:

```
class Account(){
   public int getBalance(){
      return 0;
   }
   public void trigger(){
      getBalance();
   }
}
```

Example with no `around` advice:

```
after():
   execution(public * Account.getBalance()){
      System.out.println(``After advice'');
}
```

If there is no `around` advice, then we do the following:

```
class Account(){
   public void getBalance(){}
   public void trigger(){
      getBalance();
   }
}
```

Mtd12 – `AccessModifier ReturnType ClassName.MethodName(*)`

Example:

```
public void Account.credit(*)
```

This clue tells us that the pointcut enclosing this `MethodSignature` specifies that the method must have one argument, which the pointcut does not care about its type.

We check the list of methods to be created for `ClassName`, if it has a method with, only one argument, and matches the method specifications in `MethodSignature`, we do not do anything. Otherwise, we add a new method to the class that has an `int` argument, and matches `MethodSignature`. The following is how the clue is used:

```
class Account{
   public void credit(int i1){}
   public void trigger(){
      credit(0);
   }
}
```

Mtd13 – `AccessModifier ReturnType ClassName.MethodName(*,..)`

Example:

```
public float Account.getBalance(*, ..)
```

This clue tells us that the pointcut enclosing this `MethodSignature` is interested in the method join points that have at least one argument, and that match the rest of specifications of `MethodSignature`. If there is a method with these specifications in the list of methods to be created for `ClassName` class, we do not do anything. Otherwise, we add a method that takes an `int` argument and has the rest of the specifications defined by `MethodSignature`. Note that it is a good practice to check for this clue after the previous rule: a method that satisfies the previous pointcut will also satisfy this pointcut. The following is how we use the clue, to build the constructed application code:

```
class Account{
   public float getBalance(int i1){
      return 0;
   }
   public void trigger() {
      getBalance(0);
   }
}
```

Mtd14 – `AccessModifier ReturnType ClassName.MethodName(..)`

Example:

```
public float Account.getBalance(..)
```

This clue tells us that the pointcut enclosing this `MethodSignature` does not care about the arguments of the methods join points. We check if there is a method, in the list of methods to be created for `ClassName` class, that matches the rest of the specifications of `MethodSignature`. If a match is found, we do not do anything. Otherwise, we add a new method that takes no arguments, and matches the other `MethodSignature` specifications, to the list of methods of `ClassName`. Notice that it is a good practice to check for previous rule before this one, since satisfying the previous rule will also satisfy this rule. The following is how we use this clue:

```
class Account{
   public float getBalance(){
      return 0;
   }
   public void trigger(){
      GetBalance();
   }
}
```

Mtd15 – `AccessModifier ReturnType ClassName.*(arg1,..., argN)`

Example:

```
public int Account.*(int)
```

This clue tells us that the pointcut enclosing this `MethodSignature` does not care about the method name, any method with the rest of the specifications of `MethodSignature`, is a join point for this pointcut. We check the list of the methods to be created for the `ClassName` class, and if we find a method regardless of its name but has all the other specifications of `MethodSignature`, we do not do anything. Otherwise, we create a new method we call it `mm_method#` with the same other specifications of

`MethodSignature`. The following is how this clue is used:

```
class Account{
   public int Account mm_method1(int i1){
      return 0;
   }
   public void trigger(){
     mm_method1(0);
   }
}
```

Mtd16 – `!AccessModifier * ClassName.*(arg1,..., argN)`

Example:

```
!public * Account.*(..)
```

This clue tells us that the pointcut enclosing it is not applicable to the mentioned access modifier. We check the list of methods to be created for `ClassName` class, and if at least one of them is not `public`, we do not do anything. If we do not find a method with another access modifier than the mentioned, we add a `private` method to the list of methods to be created for `ClassName` class. If the mentioned access modifier is `public`, the new method will be `private`; if it was `private` or `protected` the new method will be `public`. The clue is used as follows:

```
class Account{
   private void mm_method1();
   public void trigger(){
      mm_method1();
   }
}
```

Similarly, for access modifier `!static * ClassName.*(arg1,..., argN)`; if all of the methods are `static` we create one that is not, and `!final AccessModifier * ClassName.*(arg1,..., argN)`, if all the methods are `final` we create one that is not.

Examples:

```
  public !static * Account.*()
  public !final * Account.*()
```

The following is how the clue is used:

```
class Account{
   public void mm_method1();
   public void trigger(){
      mm_method1();
   }
}
```

Mtd17 – `* ClassName.*(..)`

Example:

```
* Account.*(..)
```

This clue tells us that the pointcut enclosing this `MethodSignature` is interested in all the member methods of the `ClassName` class, regardless of their access modifiers, names, and arguments. We check the list of methods to be created for `ClassName` class, if it is empty, we add a method called `mm_method#`, that takes no arguments, and returns `void`. If at least one method is found, we do not do anything. The following is how the clue is used:

```
class Account{
   public void Account mm_method1(){}
   public void trigger(){
     mm_method1();
   }
}
```

Mtd18 – `MethodSignature` with `!within`

Example:

```
call(public void printStatement()) && !within(Account)
    && within(!SubAccount)
```

If we do not have a method created with the specified `MethodSignature`, in a class other than `Account` and `SubAccount`, it is equivalent to creating this method in any class that we are creating, so we create it in the first class on the list of classes

to be created. If the list of classes is empty we create a new class according to the class rules, and we add the method to it. The following is how we are using this clue in the constructed application code.

```
class ClassCls1{
   public void printStatement(){}
   public void trigger(){
      ClassCls1 i1 = new ClassCls1();
       i1.printStatement();
   }
}
```

Mtd19 – **In code**

Methods can also be found in the aspect body, or inside an advice, or a method. Example:

```
aspect Aspect1{
  int x = Account.getAccountsCounter();
  Account a = new Account();
  float _balance = a.getBalance();
 }
```

In this example, we find 2 methods: `getAccountsCounter()` and `getBalance()`. We can deduce that `getAccountsCounter()` is a `static` method that takes no arguments and has an `int` return value and that the second method `getBalance()` takes no arguments, and has a `float` return type. If we have not created a method with either of these signatures before, we need to add them to the list of methods to be created for class `Account`. We create these methods with a `public` access modifier. If we cannot tell the return type of the method, like when it is called and it is not assigned to a variable, we create it with a `void` return type. We also find a constructor that is being used then we add it to the class, we explore constructors in more detail next section. The following is how the clue is transformed into the constructed application code.

```
class Account{
  public Account() {}
  public static int getAccountsCounter(){
    return 0;
```

```
5   }
6
7   public float getBalance(){
8     return 0;
9   }
10
11  public void trigger(){
12    getAccountsCounter();
13    Account i1 = new Account();
14    i1.getBalance();
15  }
16 }
```

4.6 Constructors

The same pointcuts that apply to methods also apply to constructors such as: `execution`, `call`, `withincode`, `cflow`, and `cflowbelow`. In addition, `preinitialization` and `initialization` pointcuts are used to identify constructor join points. We depend on the signature used in the pointcut to determine if the joint point is a constructor or a method. The common pointcuts are used in the same format described in Section 4.5, the only difference is the `ConstructorSignature` is plugged in instead of the `MethodSignature`. `ConstructorSignature` is used to define the intended constructor to be advised. The following is the format of the new constructor pointcuts:

```
1 initialization(ConstructorSignature)
```

The join point defined by this pointcut is any object that is created with the mentioned `ConstructorSignature`.

```
1 preinitialization(ConstructorSignature)
```

The join point defined by this pointcut is any object that is created using the mentioned `ConstructorSignature`, `preinitialization` here means before the super constructor is called.

Constructor signature takes the following format:

```
1 AccessModifier ClassName.new(arg1,...,argN) [throws ExceptionType]
```

Example:

```
public Account.new()
```

Notice that if the `ClassName` is part of the constructor signature, if this class has not been already added to the list of classes to be created, we will add it using the rules discussed in Section 4.3. The same will be done if the package has been defined in the constructor signature. E.g. : `public banking.Account.new()`. In this case we add `banking` to the list of packages to be created, and add `Account` to the list of classes to be created in the `banking` package.

If the constructor signature specifies that the constructor throws an exception, we add the `ExceptionType` to the list of exceptions to be created. In order to see the associated behavior added by the aspect, we call each defined constructor in the `trigger()` that is created for each created class. Note that a constructor is simply a special kind of method, so all what we have already described for the methods do also apply as transformation rules for constructors. In this section we focus on rules that are unique to the constructors transformation into the application code.

Ctr1 – `AccessModifier ClassName.new()`

Example:

```
withincode(public Account.new())
```

This clue tells us that `Account` class has a `public` constructor that takes no arguments. If a constructor that takes no arguments, and has the right access modifier, is not on the list of constructors to be created for class `Account`, we add one. The following is how the clue is transformed into the constructed application code:

```
class Account{
   public Account(){}
   public void trigger(){
      Account i1 = new Account();
   }
```

```
}
```

Ctr2 – `private ClassName.new(), protected ClassName.new()`

Example:

```
private Account.new()
protected Account.new()
```

This clue tells that class `Account` has a `private`/`protected` access constructor that takes no arguments. If a constructor that takes no arguments and has the right access modifier is not on the list of constructors to be created for class `Account`, we add one. Unlike methods, constructors with `private`/`protected` access modifiers are treated differently than those with `public` access modifier. In order to be able to create an instance of class `Account` to call the `trigger()` method from the `Helper` class, we add a `static` method named `createInstance()`, which is a `public` method that takes no arguments, and returns a new object of the class type. The following is how the clue is transformed into the constructed application code:

```
class Account{
   private Account() {}
   public static Account createInstance()
   {
      return new Account();
   }
   public void trigger()
   {
      Account i1 = createInstance();
   }
}
```

Ctr3 – `AccessModifier ClassName.new(arg1,...,argN) throws ExceptionName`

Example:

```
call(public Account.new(int) throws InvalidAccountNumberException)
```

The clue says that Account has a `public` constructor that takes one argument of type `int`, and throws an exception of type `InvalidAccountNumberException`. If the `Account` class does not have a constructor with the right `AccessModifier` and number and types of argument, and that throws an exception of type `InvalidAccountNumberException`, we add one to the list of constructors to be created for `Account`. The following is how the clue is used to construct the application code:

```
class Account{
   public Account(int i) throws InvalidAccountNumberException{
      if( Math.random() > 0.5)
         throw new InvalidAccountNumberException();
   }
   public void trigger(){
      try{
      Account i1 = new Account(0);
      }catch(Exception e){ }
   }
}
class InvalidAccountNumberException extends Exception{
}
```

Notice that we simulate throwing of the exception by checking if a random number is bigger than 0.5. Another thing to note is that if the access modifier is `private` or `protected`, we add `createInstance()` to the list of methods to be created for the class type. `createInstance()` has the same arguments as the constructor, and throws the same exception type.

Example:

```
call(private Account.new(int) throws InvalidAccountNumberException)
```

The following shows how `Account` class will be constructed as opposed to the `public` constructor.

```
class Account{
   private Account(int i) throws InvalidAccountNumberException{
      if( Math.random() > 0.5)
         throw new InvalidAccountNumberException();
   }
   public static Account createInstance(int i) throws
      InvalidAccountNumberException{
    return new Account(0);
   }
```

```
10    public void trigger(){
11       try {
12         Account i1 = createInstance(0);
13       } catch(Exception e){ }
14    }
15 }
```

Ctr4 – `call(ConstructorSignature1) && withincode(ConstructorSignature2)`

Example:

```
1 call(public AccountNumber.new(int)) && withincode(public Account.new())
```

This clue tells us that `Constructor1` is called from `Constructor2`. Using the constructor signatures we create the class constructors, and we add a call to `Constructor1` in `Constructor2`'s body. The following is how this clue is used in the constructed application code:

```
1  class AccountNumber{
2     public AccountNumber(int i){}
3     public void trigger(){
4        AccountNumber i1 = new AccountNumber(0);
5     }
6  }
7  class Account{
8     public Account(){
9        AccountNumber i1 = new AccountNumber(0);
10    }
11    public void trigger(){
12       Account i1 = new Account();
13    }
14 }
```

Ctr5 – `call(MethodSignature) && withincode(ConstructorSignature)`

Example:

```
1 call(public void Class1.credit(int)) && withincode(private Class2.new())
```

This clue says that `credit()` method of class `Class1`, is called in `Class2`'s `private` constructor that takes no arguments. We create a `public` method that returns `void` type, and takes no arguments named `credit()` in `Class1`. We also make sure to import

the correct package that includes the `Class1` definition before using it in `Class2`, if they are created in two different packages. We add the constructor to `Class2` according to the constructor signature, and we call the `credit()` method in `Class2` constructor, as follows:

```
class Class1{
   public void credit(int a1){}
   public void trigger(){
      credit(0);
   }
}
class Class2{
   private Class2(){
      Class1 i1 = new Class1();
      i1.credit(0);
   }
   public static Class2 createInstance()  {
      return new Class2();
   }
   public void trigger() {
   Class2 i1 = createInstance();
   }
}
```

Ctr6 – `ClassName ClassInstance = new ClassName(arg1,..,argN)`

Example:

```
try{
int number = 1234;
Account a = new Account(number);
} catch (Exception e){}
```

This clue tells us that the `Account` class has constructor that takes an `int` as argument, and throws an `Exception`. If there is no constructor defined for `Account` with these specifications, we create a `public` constructor with the specified features. The following is how this clue is used in the constructed application code.

```
class Account{
   public Account( int a1) throws Exception{
      If( Math.random() > 0.5 )
        throw new Exception();
   }
   public void trigger(){
      try{
```

```
8           Account i1 = new Account(0);
9       }
10      catch(Exception e) {}
11   }
12 }
```

CHAPTER V

Applying the Rules

5.1 Introduction

In this chapter we examine an example of how the reverse engineering rules, defined in the previous chapter, are used to construct the application code.

We begin with the aspect source code, and we work through it in several passes until there are no more ambiguities to be revealed. We start identifying packages, classes, fields, constructors, methods, and then we work on the `withincode` pointcuts. As mentioned in the previous chapter, `withincode` is a pointcut that identifies constructors and methods join points. It needs to be deferred to the end, because we need the constructors and methods to be defined first, before trying to add a call inside their bodies. In each pass we go through the entire aspect code, i.e. if the aspect code has more than one aspect, we go through the first aspect then the second aspect to identify packages, then we return to first aspect again then second aspect for classes, and same for fields, methods, constructors, and `withincode`.

5.2 Aspect Input

We start with the aspect code provided by the programmer in the file AccountAspect.aj.

```
1  import java.awt.*;
2  import javax.swing.*;
3  public aspect AccountAspect {
4
5    before():
6      preinitialization(private Account.new(int) throws Exception){
7      System.out.println("Account␣preinit␣"+thisJoinPoint);
8    }
9
10
11   pointcut publicConstructor():
12     execution(public new()) && this(Account);
13
14   after():
15     publicConstructor(){
16     System.out.println("Public␣Constructor␣"+ thisJoinPoint);
17   }
18
19
20   pointcut creditAccount(Account account, float amount):
21     call( void credit(float)) && target(account) && args(amount)
22        && within(Account);
23
24   before( Account account, float amount):
25     creditAccount(account, amount) {
26     System.out.println("Crediting␣" + amount +"␣to␣" + account );
27   }
28
29
30   before():
31     set( private float Account.balance ){
32     System.out.println("Set␣Account.balance␣" + thisJoinPoint);
33   }
34
35
36   after():
37     get( !private * Account.*){
38     System.out.println(thisJoinPoint +
39        "␣accessing␣a␣non␣private␣data␣member");
40   }
41
42
43   after() returning(float a1):
44     call(public * Account.getBalance(AccountNumber,int)) {
45     System.out.println("After␣Returning");
46   }
47
48
49   pointcut debitPointcut():
50     call(private void Account.debit(float) throws
```

```
51        InsufficientBalanceException);
52
53    after():
54      debitPointcut(){
55      System.out.println("Debit␣Pointcut");
56    }
57
58
59    pointcut methodWithincodeMethodPointcut():
60      call(public void AccountNumber.method1(int)) &&
61        withincode(private int Account.method1());
62
63    before():
64      methodWithincodeMethodPointcut(){
65      System.out.println(
66        "method␣withincode␣method␣pointcut:␣"+thisJoinPoint);
67    }
68
69
70    pointcut constructorWithincodeMethodPointcut():
71      call(public SavingsAccount.new()) &&
72        withincode(private void Account.addSavingsAccount(int));
73
74    after():
75      constructorWithincodeMethodPointcut(){
76      System.out.println(
77        "constructor␣withincode␣method␣pointcut:␣"+thisJoinPoint);
78    }
79
80
81    pointcut startOfMethodName():
82      call(public void Account.new*());
83
84    before():
85      startOfMethodName(){
86      System.out.println("matches␣start␣of␣method␣name:␣"
87        + thisJoinPoint);
88    }
89
90
91    pointcut endOfMethodName():
92      call(private void Account.*bit(..));
93
94    before():
95      endOfMethodName(){
96      System.out.println("matches␣end␣of␣method␣name:␣"
97        + thisJoinPoint);
98    }
99
100
101   pointcut startAndEndOfMethodName():
102     execution(private void add*Account(*));
103
104   after():
105     startAndEndOfMethodName(){
106     System.out.println("matches␣start␣and␣end␣of␣method␣name:␣" + thisJoinPoint);
107   }
108
```

```
109
110   pointcut anyReturnType():
111     call(public * Account.getBalance());
112
113   int around():
114     anyReturnType(){
115     System.out.println("Any␣return␣type:␣" + thisJoinPoint);
116     return proceed();
117   }
118
119
120   before():
121     call(public float Account.getBalance(*, ..)){
122     System.out.println("*␣and␣..␣arguments␣matches␣@:␣" + thisJoinPoint);
123   }
124
125
126   after():
127     call(public int Account.*(int)){
128     System.out.println("Any␣method␣name␣matches␣@:␣" + thisJoinPoint);
129   }
130
131
132   before():
133     call(public !final * Account.*()){
134     System.out.println("public␣non␣final␣matches␣@:␣" + thisJoinPoint);
135   }
136
137
138   after():
139     execution(* *(..)) && within(SavingsAccount+){
140     System.out.println("Any␣method␣SavingsAccount␣name␣matches␣@:␣"
141        + thisJoinPoint );
142   }
143
144
145   pointcut getGreenAccounts(int type):
146     call(int[][] logic.Banking.getGreenAccount(int)) && args(type);
147
148   after(int i):
149     getGreenAccounts(i){
150     System.out.println("packages:␣" + thisJoinPoint);
151   }
152
153
154   before():
155     call(void JComponent.repaint(..)){
156     System.out.println("System␣defined:" + thisJoinPoint);
157   }
158
159
160   pointcut guiPackage():
161     call( public * GUI.Layer1.*.*() );
162
163   after():
164     guiPackage(){
165     System.out.println("2␣layer␣package␣" + thisJoinPoint);
166   }
```

```
167
168
169   declare parents: SubAccount extends Account;
170
171
172   after():
173     call(private int method1()) && within(Account || SavingsAccount){
174     System.out.println("Within␣either␣class" + thisJoinPoint);
175   }
176
177
178   after():
179     call( * GUI.*.Class2.*()){
180     System.out.println("Any␣method␣in␣Class2,
181 ␣␣␣␣␣␣Class2␣belongs␣to␣a␣sub␣package␣of␣package␣GUI");
182   }
183
184
185   after():
186     execution( public * DataBase.*.Connection.getConnection() ){
187     System.out.println("Any␣sub␣package␣in␣DataBase␣package");
188   }
189
190
191   before():
192     call( public void printStatement()) && !within(Account)
193         && within(!SubAccount) {
194       System.out.println(
195         "!within(Account)␣and␣within(!SubAccount)"
196           + thisJoinPoint);
197   }
198
199
200   after():
201     call(public * logic.SubLogic.*.*()){
202     System.out.println("Create␣a␣new␣Class");
203   }
204
205   pointcut exceptionHandler():
206     handler(InsufficientBalanceException);
207
208   after():
209     exceptionHandler(){
210     System.out.println("Exception␣Handler␣matches␣@:␣" + thisJoinPoint);
211   }
212
213   int Account.x = 0;
214   before():
215     set( int Account.x ){
216     System.out.println("Set␣Account.x␣ITD␣" + thisJoinPoint);
217   }
218
219
220   void aspectMethod()
221   {
222     Account.print("My␣String");
223     RunnableWithReturn worker = new RunnableWithReturn();
224     try{
```

```
225         EventQueue.invokeAndWait(worker);
226       } catch (Exception e){}
227     }
228
229 }
```

5.3 Identifying the Elements

5.3.1 Identifying Packages

We first start looking for packages.

```
1 import java.awt.*;
2 import javax.swing.*;
```

At line 1 and 2 we find packages `java.awt.*` and `javax.swing`, but since they are system-defined packages we do not need to create them in the constructed application code (Pkg1).

```
145   pointcut getGreenAccounts(int type):
146     call(int[][] logic.Banking.getGreenAccount(int)) && args(type);
```

In pointcut `getGreenAccounts`, in line 146 package `logic` is mentioned, since `logic` is not a system-defined package and we do not have a package called `logic` in the list of packages to be created. We add `logic` to the list of packages to be created (Pkg1).

```
160   pointcut guiPackage():
161     call( public * GUI.Layer1.*.*() );
```

At line 161, inside the `guiPackage` pointcut, `GUI.Layer1` is mentioned, since `GUI` is not system-defined, and we do not have a `GUI` package already created, we add `GUI` to the list of packages to be created. We add `Layer1` as a sub package of `GUI` (Pkg2).

```
178   after():
179     call( * GUI.*.Class2.*()){
180     System.out.println("Any␣method␣in␣Class2,
181 ␣␣␣␣␣␣Class2␣belongs␣to␣a␣sub␣package␣of␣package␣GUI");
182   }
```

At line 179, in the after advice, `GUI.*` is mentioned, since `GUI` is already in the list of packages to be created, we do not need to add it. And since `GUI` has a sub package, `Layer1` in this example, then we do not need to create another one (Pkg4).

```
185   after():
186     execution( public * DataBase.*.Connection.getConnection() ){
187     System.out.println("Any␣sub␣package␣in␣DataBase␣package");
188   }
```

In line 186, we see `DataBase.*` package, we add `DataBase` to the list of packages to be created. `*` referrers to any sub package of `DataBase`. There are none defined so far, so we continue this pass identifying packages, a sub package of `DataBase` may be defined later in the aspect code.

```
200   after():
201     call(public * logic.SubLogic.*.*()){
202     System.out.println("Create␣a␣new␣Class");
203   }
```

Line 201 in the `after` advice, we find `logic.SubLogic`, since `logic` package exists, we search it for a sub package with name `SubLogic`, since we do not find one, we add `SubLogic` as a sub package of package `logic` (Pkg2).

When we reach the end of file of the aspect source code, if there are no package ambiguities we start a new pass for identifying classes. But since we still have a package ambiguity to be resolved, we need to start another pass for revealing the ambiguous package.

```
185   after():
186     execution( public * DataBase.*.Connection.getConnection() ){
187     System.out.println("Any␣sub␣package␣in␣DataBase␣package");
188   }
```

So we start and we get to `DataBase.*` again, since there are no sub packages inside `DataBase` package, we need to create one, we call it `package_p1` (Pkg4). We have a counter that we increment whenever we create an element that does not have a name, and this number becomes a part of the element's name.

Since there are no more packages ambiguities to be resolved, we move on to the next element. We start a new pass for identifying classes.

By the end of the package passes we have identified the following packages:

```
DataBase
  package_p1
GUI
  Layer1
logic
  SubLogic
```

5.3.2 Identifying Classes

```
5   before():
6     preinitialization(private Account.new(int) throws Exception){
7     System.out.println("Account preinit "+thisJoinPoint);
8   }
```

We start from the beginning again, at line 6 we find `Account`, in the first unnamed pointcut. Since `Account` is not a system-defined class, and it has not been added before to the list of classes to be created. `Account` is not fully qualified, so we add it to the default package (Cls1). At line 6 also, we find `int`, but since `int` is system-defined type we do not need to create it (Cls1). Also at line 6 we see `Exception`, but since it is a system-defined type we do not add it to the list of exceptions to be created (Cls9).

```
11  pointcut publicConstructor():
12    execution(public new()) && this(Account);
```

In pointcut publicConstructor at line 12, we see `Account` class, since it is already added to the list of classes to be created, we do not add it again (Cls7).

```
20  pointcut creditAccount(Account account, float amount):
```

```
21      call( void credit(float)) && target(account) && args(amount)
22        && within(Account);
23
24    before(Account account, float amount):
25        creditAccount(account, amount) {
26      System.out.println("Crediting␣" + amount +"␣to␣" + account );
27    }
```

In pointcut `creditAccount`, we see `Account` and `float` as arguments of the pointcut, we follow (Cls1), not creating anything since `Account` is already in the list of classes to be created, and `float` is a system-defined type. We also see `void` as return type of `credit()` method, and `float` as its argument, and we also do not create them since they are system-defined types. We also find `target(account)`, and `args(amount)` which are of types `Account` and `float` respectively, following Cls7 we do not create them. And lastly we find `within(Account)` and following Cls1 we do not need to create `Account` again. In the `before` advice at line 24, we find `Account` and `float` again and we do not create them according to Cls1.

```
30    before():
31      set( private float Account.balance ){
32      System.out.println("Set␣Account.balance␣" + thisJoinPoint);
33    }
34
35
36    after():
37      get( !private * Account.*){
38      System.out.println(thisJoinPoint +
39        "␣accessing␣a␣non␣private␣data␣member");
40    }
```

In the unnamed pointcuts at line 31 and 37, we find `Account` again and we follow Cls1 not creating `Account`.

```
43    after() returning(float a1):
44      call(public * Account.getBalance(AccountNumber,int)) {
45      System.out.println("After␣Returning");
46  }
```

The `after returning` advice at line 43, has an argument of type `float`, and according to Cls11, we do not need to add `float` to the list of classes to be created, since it is a system-defined type. We do find `Account` and following Cls1 we do not

create it. We then find `AccountNumber` as an argument of getBalance and following Cls1 we add it to the list of classes to be created. Since `AccountNumber` is not a fully qualified name, we add `AccountNumber` to be created in the default package. We also find `int` as the second argument of getBalance, but according to Cls1 we do not create it.

```
49  pointcut debitPointcut():
50    call(private void Account.debit(float) throws
51      InsufficientBalanceException);
```

In pointcut `debitPointcut` we see `void`, `Account`, and `float` and we do not create anything (Cls1). We see `InsufficientBalanceException`, so we follow Cls9. `InsufficientBalanceException` is not system-defined, and it is not in the list of exceptions to be created, so we add it. Since `InsufficientBalanceException` is not a fully qualified name, we create `InsufficientBalanceException` in the default package.

```
59  pointcut methodWithincodeMethodPointcut():
60    call(public void AccountNumber.method1(int)) &&
61      withincode(private int Account.method1());
```

In pointcut `methodWithincodeMethodPointcut`, we find `void`, `AccountNumber`, `int`, and `Account`, and we do not add them according to Cls1.

```
70  pointcut constructorWithincodeMethodPointcut():
71    call(public SavingsAccount.new()) &&
72      withincode(private void Account.addSavingsAccount(int));
```

In pointcut `constructorWithincodeMethodPointcut`, we find `SavingsAccount` and we add it to the list of classes to be created in the default package, also according to Cls1. We also find `void`, `Account`, and `int`, and we do not do anything with them.

```
81  pointcut startOfMethodName():
82    call(public void Account.new*());


91  pointcut endOfMethodName():
92    call(private void Account.*bit(..));


110 pointcut anyReturnType():
```

```
111     call(public * Account.getBalance());


120   before():
121     call(public float Account.getBalance(*, ..)){
122     System.out.println("* and ... arguments matches @: " + thisJoinPoint);
123   }


126   after():
127     call(public int Account.*(int)){
128     System.out.println("Any method name matches @: " + thisJoinPoint);
129   }


132 before():
133     call(public !final * Account.*()){
134     System.out.println("public non final matches @: " + thisJoinPoint);
135   }
```

In line 82 and 92 we find both `void`, and `Account`, and in line 111 we find `Account`, and in line 121 we find `float` and `Account`, also in line 127 we find `int` and `Account`, and in line 133 we find `Account`, so we apply Cls1, not creating anything.

```
138   after():
139     execution(* *(..)) && within(SavingsAccount+){
140     System.out.println("Any method SavingsAccount name matches @: " +
141       thisJoinPoint );
142   }
```

In line 139 we find `SavingsAccount+`, so according to Cls6 we should treat that as `SavingsAccount`, which is already added to the list of classes to be created, so we do not need to add it again.

```
145   pointcut getGreenAccounts(int type):
146     call(int[][] logic.Banking.getGreenAccount(int)) && args(type);
147
148   after(int i):
149     getGreenAccounts(i){
150     System.out.println("packages: " + thisJoinPoint);
151   }
```

In pointcut `getGreenAccounts`, we find `int` and we do not create it (Cls1). We also find `logic.Banking`, which is not a system-defined class. We have the fully qualified name for `Banking`, and we know it is in `logic` package, so we search the list of classes for `Banking` class that has `logic` as its package. Since there is no class created with

these specifications, we create a new class called `Banking` with package set to `logic` (Cls1). We also see `args(type)` which is of type `int`, and we see it again as argument of the `after` advice at line 148 so we do not create it (Cls1).

```
154   before():
155     call(void JComponent.repaint(..)){
156     System.out.println("System␣defined:" + thisJoinPoint);
157   }
```

At line 155 in the `before` advice, we find `JComponent`, which is a system-defined class whose package is appropriately imported with the aspect code, we do not create `JComponent` according to (Cls1).

```
160   pointcut guiPackage():
161     call( public * GUI.Layer1.*.*() );
```

At pointcut `guiPackage`, we find `GUI.Layer1.*`. Since there are no classes defined in `GUI.Layer1` we skip this pattern for now, and we continue looking for classes, we might find defined in this package later in the aspect code.

```
169   declare parents: SubAccount extends Account;
```

In `declare parents` line 169 we find a new class `SubAccount`, so we add it to the list of classes to be created in the default package (Cls13). We also find class `Account` and we do not do anything with it (Cls1).

```
172   after():
173     call(private int method1()) && within(Account || SavingsAccount){
174     System.out.println("Within␣either␣class" + thisJoinPoint);
175   }
```

In the `after` advice at line 173, we find type `int` and we do not do anything with it. We also find `within(Account || SavingsAccount)` but since both of them are already defined, we do not do anything (Cls2).

```
178   after():
179     call( * GUI.*.Class2.*()){
180     System.out.println("Any␣method␣in␣Class2,
181 ␣␣␣␣␣␣Class2␣belongs␣to␣a␣sub␣package␣of␣package␣GUI");
182   }
```

In line 179 in the `after` advice, we find `GUI.*.Class2`, since at this point all the packages have being identified, we search `GUI` and we only find `Layer1` as its only subpackage. And since `Class2` is not defined in `Layer1`, we add `Class2` to the list of classes to be defined, whose package is `GUI.Layer1`. Note: if `GUI`, had multiple sub packages, defining `Class2` is equivalent if done in any. But for consistency, we would have add `Class2` to the first sub package defined in `GUI` (Cls1).

```
185  after():
186    execution( public * DataBase.*.Connection.getConnection() ){
187    System.out.println("Any␣sub␣package␣in␣DataBase␣package");
188  }
```

Same with the `after` advice at line 186, we have `DataBase.*.Connection`, we add class `Connection` to the list of classes to be created in `DataBase.package_p1` (Cls1).

```
191  before():
192    call( public void printStatement()) && !within(Account) &&
193        within(!SubAccount) {
194      System.out.println("!within(Account)␣and␣within(!SubAccount)"
195        + thisJoinPoint);
196  }
```

In the `before` advice, at line 192 we find `!within(Account) && (!SubAccount)`, since the application we have other classes in the list of classes to be created we do not need to do anything (Cls15) and (Cls16).

```
200  after():
201    call(public * logic.SubLogic.*.*()){
202    System.out.println("Create␣a␣new␣Class");
203  }
```

In the `after` advice at line 201, we find `logic.SubLogic.*`, since no classes are defined in the `logic.SubLogic` package, we skip this pattern for now, the class may be defined later in the aspect code.

```
205  pointcut exceptionHandler():
206    handler(InsufficientBalanceException);
```

In pointcut `exceptionHandler`, we find `InsufficientBalanceException`, since it is already added to the list of exceptions to be created, we do not do anything.

```
213  int Account.x = 0;
214  before():
215    set( int Account.x ){
216    System.out.println("Set␣Account.x␣ITD␣" + thisJoinPoint);
217  }
```

In lines 213 and 215 we find `int` and `Account` types and we do not do anything according to (Cls1).

```
220  void aspectMethod()
221  {
222    Account.print("My␣String");
223    RunnableWithReturn worker = new RunnableWithReturn();
224    try{
225      EventQueue.invokeAndWait(worker);
226    } catch (Exception e){}
227  }
```

And in line 222 we find `Account` and we do not do anything according to (Cls1).

In line 223 we find `RunnableWithReturn`, and we add it to the list of classes to be created.

In line 225 we find `EventQueue`, and since the appropriate package is imported we are able to determine that it is system-defined. So we do not do anything. Using Reflection API, we can tell that `invokeAndWait()` is a system-defined method that takes an argument of type `Runnable`. Since the variable passed to this function is not of type `Runnable`, it must be a child of it. Since `Runnable` is an interface, the variable type, i.e. `RunnableWithReturn` must implement `Runnable`, and its abstract methods, `run()` in this case (Cls17).

```
160  pointcut guiPackage():
161    call( public * GUI.Layer1.*.*() );
```

Since there are still some classes ambiguities, we start a new pass, to resolve what we skipped in the previous one. We find `GUI.Layer1.*` and at this point we know that there is a class named `Class2` in this package, so `Class2` satisfies this pattern.

```
200   after():
201     call(public * logic.SubLogic.*.*()){
202     System.out.println("Create␣a␣new␣Class");
203   }
```

The other ambiguity is `logic.SubLogic.*` and since there are no classes defined in this package, we need to define one, so we add `ClassCls2` to the list of classes to be created in the `logic.SubLogic` package.

By the end of this passes we have the following elements are identified:

```
In default package:
  Account
  AccountNumber
  InsufficientBalanceException
  RunnableWithReturn and its method public void run()
  SavingsAccount
  SubAccount
In DataBase.package_p1 package:
  Connection
In GUI.Layer1 package:
  Class2
In logic package:
  Banking
In logic.SubLogic package:
  ClassCls2
```

Since we are done identifying the classes, it is time to start identifying the class elements.

5.3.3 Identifying Fields

We start with the fields

```
30  before():
31    set( private float Account.balance ){
32    System.out.println("Set␣Account.balance␣" + thisJoinPoint);
33  }
```

In line 31, we find `private float Account.balance` (Fld1, we add the `private float balance` field member to the list of fields to be created for the `Account` class, and we flagged as a `set` field so we remember to assign a value to it in the `trigger()` function.

```
36  after():
37    get( !private * Account.*){
38    System.out.println(thisJoinPoint + "␣accessing␣a␣non␣private␣data␣member");
39  }
```

In line 37, we found a `get` for `!private * Account.*`, since all the fields created so far for `Account` are `private`, we skip this pattern for now, we might find an `Account` field that is not `private` later in the aspect code.

```
213  int Account.x = 0;
214  before():
215    set( int Account.x ){
216    System.out.println("Set␣Account.x␣ITD␣" + thisJoinPoint);
217  }
```

At line 213 and 215 we find `int Account.x`, but we ignore it as it is an intertype member declaration, since it is added by the aspect itself and not part of the original code application. So, we also do not need it to be part of the constructed code application. The code added by the aspect to the original application, will still be woven with the constructed application, giving us the same behavior, it added to the original application.

Since there still are some ambiguities for the fields, we go another pass through the ambiguities. In our example, the ambiguity is `get(!private * Account.*)`. Since

all the fields of identified for `Account` are `private`, we add a new field `public f3` of type `int` and we flag it as get, so we remember to get its value in trigger method (Fld3) and (Fld4).

Since we are done identifying the fields we move on to identifying the methods. By the end of these passes we the following elements identified:

```
For class Account:
  private float balance;
  public int f3;
```

5.3.4 Identifying methods

```
20  pointcut creditAccount(Account account, float amount):
21    call( void credit(float)) && target(account) && args(amount)
22      && within(Account);
```

In pointcut `creditAccount`, we find `void credit(float)` and `within(Account)`, since `Account` does not have this method in its list of methods, we follow (Mtd4) adding a method called `credit()` that has one argument of type `float` and returns a `void` type, to the list of methods to be created for class Account.

```
43  after() returning(float a1):
44    call(public * Account.getBalance(AccountNumber,int)) {
45    System.out.println("After␣Returning");
46  }
```

At line 44, we find a call to a `getBalance()` method, and since we do not find this method in the list of `Account` methods, we follow (Mtd11) to add a `public` method called `getBalance()`, that takes 2 arguments of types `AccountNumber` and `int`, and returns `float` type, to the list of methods of `Account`.

```
49  pointcut debitPointcut():
50    call(private void Account.debit(float)
51      throws InsufficientBalanceException);
```

In `debitPointcut` at line 50, we find the `debit()` method, and since `Account` does not have this method in its list of methods to be created, we follow (Mtd3) to add a `private` method called `debit()`, that takes one argument of type `float`, and returns a `void` type, and throws an exception of type `InsufficientBalanceException`.

```
59  pointcut methodWithincodeMethodPointcut():
60    call(public void AccountNumber.method1(int)) &&
61      withincode(private int Account.method1());
```

In pointcut `methodWithincodeMethodPointcut` we see a call to `method1()`, and since `AccountNumber` has no methods of the signature of `method1`, and `Account` does not have a method with `method1` signature respectively, we follow Mtd1 to add a `public` method called `method1` that takes one argument of type `int` and returns a `void` type to the class `AccountNumber`; and add a `private` method called `method1()` that takes no arguments and returns `int` type to the `Account` class. Note that we will visit this pointcut again to resolve the `withincode` later.

```
70  pointcut constructorWithincodeMethodPointcut():
71    call(public SavingsAccount.new()) &&
72      withincode(private void Account.addSavingsAccount(int));
```

In pointcut `constructorWithincodeMethodPointcut`, we find a call to `addSavingsAccount(int)`, and since `Account` does not have a method with `addSavingsAccount` signature, we to the list of methods to be created for `Account`, a `private` method called `addSavingsAccount` that takes one argument of type `int`, and returns a `void` type (Mtd1).

```
81  pointcut startOfMethodName():
82    call(public void Account.new*());
```

In pointcut `startOfMethodName`, we find `public void Account.new*()`. Since `Account` does not have a method that starts with `new` and have this signature, we skip it for now, we may find it later in the aspect code.

```
91  pointcut endOfMethodName():
92    call(private void Account.*bit(..));
```

In pointcut `endOfMethodName`, we find `private void Account.*bit(..)`, this pointcut matches `Account`'s `private void debit(float)` that was created previously, so we do not need to add any new methods (Mtd9).

```
101  pointcut startAndEndOfMethodName():
102    execution(private void add*Account(*));
```

In pointcut `startAndEndOfMethodName`, we find `private void add*Account(*)`, and since the class `Account` has a `private` method called `addSavingsAccount` and returns a `void` type, this method will satisfy this pointcut, so there is no need to add a new method (Mtd10).

```
110  pointcut anyReturnType():
111    call(public * Account.getBalance());
```

In pointcut `anyReturnType`, at line 111 we find `public * Account.getBalance()`. `Account` does not have a `public getBalance()` method that takes no arguments, but it is used in an `around` advice that returns an `int`. So, we add a `public` method called `getBalance()`, that takes no arguments, and returns `int` type to the list of methods to be created for class `Account` (Mtd11).

```
120  before():
121    call(public float Account.getBalance(*, ..)){
122    System.out.println("* and ... arguments matches @: " + thisJoinPoint);
123  }
```

At line 112, we find `public float Account.getBalance(*, ..)`. This pointcut is basically interested in a `public getBalance()` method of class `Account` that takes at least one argument, and returns a `float` type. `Account` already has a method of signature `public float getBalance(AccountNumber, int)` that would satisfy this pointcut, so we do not need to add any new methods (Mtd13).

```
126  after():
127    call(public int Account.*(int)){
128    System.out.println("Any method name matches @: " + thisJoinPoint);
129  }
```

At line 127, we find `public int Account.*(int)`, since `Account` does not have a `public` method that only takes one argument of type `int`, and has a return value of type `int`, we skip it for now, we may find a method that satisfy it later in the aspect code.

```
132  before():
133    call(public !final * Account.*()){
134    System.out.println("public non final matches @: " + thisJoinPoint);
135  }
```

At line 133, we find `public !final * Account.*()`. Since `Account` does not have a `public` method that takes no arguments yet, we are going to skip this for now, we may find a method that satisfies it later in the aspect code.

```
138  after():
139    execution(* *(..)) && within(SavingsAccount+){
140    System.out.println("Any method SavingsAccount name matches @: " +
141      thisJoinPoint );
142  }
```

At line 139, we find `execution(* *(..)) && within(SavingsAccount+)`, which will be satisfied by any method in the `SavingsAccount` class. Since `SavingsAccount` has no methods defined yet, we skip this pointcut for now. We might find the method that satisfies it later in the aspect code.

```
145  pointcut getGreenAccounts(int type):
146    call(int[][] logic.Banking.getGreenAccount(int)) && args(type);
```

At line 146, we find `int[][] logic.Banking.getGreenAccount(int)`, we follow (Mtd1) to add a method to class `Banking` in `logic` package, called `getGreenAccount()` that takes one argument of type `int` and returns a 2 dimensional array of `int`.

```
154  before():
155    call(void JComponent.repaint(..)){
156    System.out.println("System defined:" + thisJoinPoint);
157  }
```

At line 155, we find void `JComponent.repaint(..)`, since it is system-defined we do not need to recreate it.

```
160   pointcut guiPackage():
161     call( public * GUI.Layer1.*.*() );
```

At line 161, we find public * GUI.Layer1.*.*(). Since Class2 has no methods defined yet, we skip this for now, we may find a method that satisfies it later in the aspect code.

```
172   after():
173     call(private int method1()) && within(Account || SavingsAccount){
174     System.out.println("Within␣either␣class" + thisJoinPoint);
```

At line 173, we find call(private int method1()) && within(Account ||SavingsAccount), since Account has private int method1(), we do not need to do anything.

```
178   after():
179     call( * GUI.*.Class2.*()){
180     System.out.println("Any␣method␣in␣Class2,
181 ␣␣␣␣␣␣Class2␣belongs␣to␣a␣sub␣package␣of␣package␣GUI");
182   }
```

At line 179, we find call(* GUI.*.Class2.*()). Since Class2 still does not have any methods defined, we skip this pointcut for now, we may find a method that satisfies it later in the aspect code.

```
185   after():
186     execution( public * DataBase.*.Connection.getConnection() ){
187     System.out.println("Any␣sub␣package␣in␣DataBase␣package");
188   }
```

At line 186, we find execution(public * DataBase.*.Connection.getConnection()), since we do not know the return type of the method, we skip this pointcut for now, we may find a method that satisfies it later in the aspect code.

```
191   before(): call( public void printStatement()) && !within(Account) &&
192       within(!SubAccount) {
193     System.out.println("!within(Account)␣and␣within(!SubAccount)" +
194       thisJoinPoint);
195   }
```

At line 192, we find call(public void printStatement()) && !within(Account) && within(!SubAccount). We can define printStatement() in any class other than Ac-

count or SubAccount, so we choose AccountNumber. We add a public method called printStatement(), that takes no arguments, and returns void type in the list of methods to be created for AccountNumber class (Mtd18).

```
200   after():
201     call(public * logic.SubLogic.*.*()){
202     System.out.println("Create␣a␣new␣Class");
203   }
```

At line 201, we find call(public * logic.SubLogic.*.*()), since ClassCls2 in logic.SubLogic, does not have any methods defined yet, we skip this pointcut for now, we may find a method that satisfies it later in the aspect code.

```
220   void aspectMethod()
221   {
222     Account.print("My␣String");
223     RunnableWithReturn worker = new RunnableWithReturn();
224     try{
225       EventQueue.invokeAndWait(worker);
226     } catch (Exception e){}
227   }
```

At line 222 we find Account.print("My String"). We search the list of methods to be created for class Account, for a static method called print, that takes one argument of type String. If we find one, we do not do anything. Otherwise we add to the list of methods to be created for class Account, a public static method called print(), that takes one argument of type String and returns void type.

At line 225 we find EventQueue.invokeAndWait(worker). Since it is a system-defined method, we do not do anything.

If there are no methods ambiguities to be resolved we can go to next pass for identifying constructors, but since there are still some methods ambiguities to be revealed we do another pass for methods.

```
81   pointcut startOfMethodName():
82     call(public void Account.new*());
```

We start with the first method ambiguity at line 82, we search our list of methods for a `public` method which name starts with `new`, and that takes no arguments, and returns a `void` type. Since there is no method defined with these specifications, we add a new method called `new_m4()` with these specifications to the list of methods to be created for class `Account` (Mtd8).

```
126  after():
127    call(public int Account.*(int)){
128    System.out.println("Any␣method␣name␣matches␣@:␣" + thisJoinPoint);
129  }
```

At line 127 we see `call(public int Account.*(int))`. Since there is no `public` method in class `Account` that take one argument of type `int` and returns `int` type. We add a method called `mm_method5()`, with the provided specifications to the list of methods of class `Account` (Mtd15).

```
132  before():
133    call(public !final * Account.*()){
134    System.out.println("public␣non␣final␣matches␣@:␣" + thisJoinPoint);
135  }
```

We encounter `call(public !final * Account.*())` at line 133. Since there is at least one `public` method that is not `final` in class `Account`, we do not do anything (Mtd16).

```
138  after():
139    execution(* *(..)) && within(SavingsAccount+){
140    System.out.println("Any␣method␣SavingsAccount␣name␣matches␣@:␣" +
141      thisJoinPoint );
142  }
```

At line 139 we find `execution(* *(..)) && within(SavingsAccount+)`. Since there are no methods defined for class `SavingsAccount`, we create a new `public` method called `mm_method6()` that takes no arguments and returns `void` type, to the list of methods to be created for class `SavingsAccount` (Mtd15).

```
160  pointcut guiPackage():
161    call( public * GUI.Layer1.*.*() );
```

At line 161 we find `call(public * GUI.Layer1.*.*())`. Since `GUI.Layer1.Class2` has a `public method2()`, it will satisfy this pointcut.

```
178   after():
179     call( * GUI.*.Class2.*()){
180     System.out.println("Any␣method␣in␣Class2,
181 ␣␣␣␣␣␣Class2␣belongs␣to␣a␣sub␣package␣of␣package␣GUI");
182   }
```

At line 179 we find `call(* GUI.*.Class2.*())`. This pointcut also will be satisfied with `public void GUI.Layer1.Class2.method2()`.

```
185   after():
186     execution( public * DataBase.*.Connection.getConnection() ){
187     System.out.println("Any␣sub␣package␣in␣DataBase␣package");
188   }
```

At line 186 we find `execution(public * DataBase.*.Connection.getConnection())`. Since there are no `public` methods defined in `DataBase.package_p1.Connection` whose name is `getConnection()`, we add a new `public` method that is called `getConnection()` that takes no arguments, and returns `void` type to the list of methods to be created for `DataBase.package_p1.Connection` class (Mtd11).

```
200   after():
201     call(public * logic.SubLogic.*.*()){
202     System.out.println("Create␣a␣new␣Class");
203   }
```

At line 201 we find `call(public * logic.SubLogic.*.*())`. Since there are no methods defined in `ClassCls2`, we define a new `public` method called `mm_method7()` that takes no arguments and returns `void` type, to the list of methods to be created for `ClassCls2` class (Mtd15).

By the end of these passes we have the following elements identified:

```
For class Account:

  public void credit(float a1)

  public float getBalance(AccountNumber a1, int a2)
```

```
  private void debit (float a1) throws InsufficientBalanceException
  private int method1()
  private void addSavingsAccount(int a1)
  public void new_m4()
  public int getBalance()
  public int mm_method5(int a1)
  public static void print(String a1)

For class AccountNumber:
  public void method1(int a1)
  public void printStatement()

For class SavingsAccount:
  void mm_method6()

For class DataBase.package_p1.Connection:
  public void getConnection()

For class GUI.Layer1.Class2:
  public void method2()

For class logic.Banking:
  public int[][] getGreenAccount(int a1)

For class logic.SubLogic.ClassCls2:
  public void mm_method7()
```

Now we are done identifying all the methods, we can start identifying the constructors.

5.3.5 Identifying Constructors

```
5   before():
6     preinitialization(private Account.new(int) throws Exception){
7     System.out.println("Account preinit "+thisJoinPoint);
8   }
```

At line 6 we find `preinitialization(private Account.new(int) throws Exception)`, we create a `private` constructor for class `Account` that takes one argument of type `int`, and throws `Exception`. We also create a `public static` method called `getInstance()` that takes the same arguments and throws the same `Exception`, and returns a new `Account` object. The purpose of the `getInstance()` method, as mentioned in the previous chapter, is to be able to create an object of the `Account` class from outside the class (Ctr3).

```
11  pointcut publicConstructor():
12    execution(public new()) && this(Account);
```

At line 12 we find `execution(public new()) && this(Account)`. Since we do not have a `public` constructor that takes no arguments for class `Account`, we add one to the list of constructors to be created for `Account` (Ctr1).

```
70  pointcut constructorWithincodeMethodPointcut():
71    call(public SavingsAccount.new()) &&
72     withincode(private void Account.addSavingsAccount(int));
```

At line 71 we find `call(public SavingsAccount.new())`. Since `SavingsAccount` does not have a `public` constructor that takes no arguments defined, we add one to it (Ctr1).

```
220 void aspectMethod() {
221   Account.print("My String");
222   RunnableWithReturn worker = new RunnableWithReturn();
223   try{
224     EventQueue.invokeAndWait(worker);
225   } catch (Exception e){}
```

```
226  }
```

At line 223 we find `RunnableWithReturn worker = new RunnableWithReturn()`. We add a `public` constructor that takes no arguments to the `RunnableWithReturn` class (Mtd19).

By the end of this pass, we have identified the following elements:

```
For class Account:
  private Account(int) throws Exception
  public static getInstance(int) throws Exception
  public Account()

For class SavingsAccount:
  public SavingsAccount()
```

Since there are no ambiguities to be resolved for constructors, we can go ahead with our next pass, going through the `withincode` pointcuts.

5.3.6 `withincode` Pass

Since we now have both the constructors and the methods of the constructed application defined, we can look at the `withincode` pointcuts, to make sure we include the appropriate calls in the constructors and methods bodies.

```
59  pointcut methodWithincodeMethodPointcut():
60    call(public void AccountNumber.method1(int)) &&
61      withincode(private int Account.method1());
```

At lines 60 and 61 we find `call(public void AccountNumber.method1(int)) && withincode (private int Account.method1())`. So, in the body of `private int Account.method1()`, we need to create an instance of `AccountNumber`, and use it to call `AccountNumber.method1(int)` (Mtd6). We create an object of `AccountNumber` using the first `public` constructor in the

list of constructors of `AccountNumber` class. If there are no `public` constructors and there are `private` constructors, we use the first `getInstance` method to create the object. We did not define any constructors for `AccountNumber`, since none of them is mentioned explicitly in the aspect code. So we use the default system-defined constructor to create an `AccountNumber` object. The following is how `private int Account.method1()` will look like.

```
1   private int method1()
2   {
3     AccountNumber i1 = new AccountNumber();
4     i1.method1(0);
5     return 0;
6   }
```

```
70  pointcut constructorWithincodeMethodPointcut():
71    call(public SavingsAccount.new()) &&
72      withincode(private void Account.addSavingsAccount(int));
```

At lines 71 and 72 we find `call(public SavingsAccount.new()) && withincode(private void Account.addSavingsAccount(int))`. So in the body of `private void Account.addSavings-Account(int)`, we need to create an instance of `SavingsAccount` (Mtd7). We create an object of `SavingsAccount` using the constructor mentioned, i.e. `public` constructor that takes no arguments. The following is how

```
1   private void Account.addSavingsAccount(int) will look like.
2   private void addSavingsAccount(int a1)
3   {
4     SavingsAccount i1 = new SavingsAccount();
5   }
```

By the end of this pass we have defined more specifications into the body of the following elements:

`private int Account.method1()`

`private void Account.addSavingsAccount(int a1)`

5.4 The Output

5.4.1 Application Classes

Once we have all the elements identified, we flesh out the application with trigger functions added.

Account class:

```
1  public class Account {
2
3    private float balance;
4
5    public int f3;
6
7    private Account(int a1) throws Exception{
8      if( Math.random() > 0.5 )
9        throw new Exception();
10   }
11
12   public static Account createInstance(int a1) throws Exception
13   {
14     return new Account(0);
15   }
16
17   public Account() {}
18
19   public void credit(float a1){}
20
21   public float getBalance(AccountNumber a1, int a2)
22   {
23     return 0;
24   }
25
26   private void debit (float a1) throws InsufficientBalanceException{
27     if( Math.random() > 0.5 )
28       throw new InsufficientBalanceException();
29   }
30
31   private int method1()
32   {
33     AccountNumber i1 = new AccountNumber();
34     i1.method1(0);
35     return 0;
36   }
37
38   private void addSavingsAccount(int a1)
39   {
40     SavingsAccount i1 = new SavingsAccount();
41   }
42
43   public void new_m4(){}
44
```

```
45    public int getBalance(){
46      return 0;
47    }
48
49    public int mm_method5(int a1){
50      return 0;
51    }
52
53    public static void print(String a1)
54    {
55    }
56
57    public void trigger()
58    {
59      print("");
60      try{
61        Account i1 = createInstance(0);
62        i1.balance = 0;
63        int i2 = i1.f3;
64        i1.credit(0);
65        i1.getBalance(new AccountNumber(), 0);
66        i1.method1();
67        i1.addSavingsAccount(0);
68        i1.new_m4();
69        i1.getBalance();
70        i1.mm_method5(0);
71        try{
72          i1.debit(0);
73        }catch(Exception e){}
74      }catch(Exception e){}
75
76
77      Account i3 = new Account();
78      i3.balance = 0;
79      int i4 = i3.f3;
80      i3.credit(0);
81      i3.getBalance(new AccountNumber(), 0);
82      i3.method1();
83      i3.addSavingsAccount(0);
84      i3.new_m4();
85      i3.getBalance();
86      i3.mm_method5(0);
87      try{
88        i3.debit(0);
89      }catch(Exception e){}
90
91    }
92  }
```

Note that we return the default values of the types for the return types of the functions. If a constructor or a method throws an exception, we simulate the checking if a random number is less than 0.5. As you can see, the `trigger()` function is added. We start by calling the `static` functions, then creating object by object,

and accessing its fields. Note that `balance` is being set, and `f3` is being get, according to their flags. Then we call all the functions with their default values. Since `debit()` throws an exception it should be called in a `try ... catch` block. Also note that we create an instance of the class with each constructor and call all the methods, and access all the fields for each instance created. This way allows us to cover all the possibilities intended by the aspect developer. If the method or the field are declared static, then we use them directly without the instance in the `trigger()` function. If the class does not have an explicit constructor, and have methods or fields that need to be exercised in the `trigger()` function, we use the system-default constructor to create the instance.

And we do the same thing for all the classes.

```
AccountNumber class:
public class AccountNumber {

  public void method1(int a1){
  }

  public void printStatement(){

  }

  public void trigger(){
    AccountNumber i1 = new AccountNumber();
    i1.method1(0);
    i1.printStatement();
  }
}

InsufficientBalanceException exception:
public class InsufficientBalanceException extends Exception{

}
```

Note that no `trigger()` method is added to the exception, since no methods were created for it.

RunnableWithReturn class:

```
public class RunnableWithReturn implements Runnable{

  public RunnableWithReturn(){}

```

```
5   public void run() {
6
7   }
8
9   public void trigger()
10  {
11    RunnableWithReturn i1 = new RunnableWithReturn();
12  }
13 }
```

Note that `run()` method is not called in `trigger()` method, since it is not explicitly mentioned in the aspect.

SavingsAccount class:

```
1 public class SavingsAccount {
2
3   public SavingsAccount(){}
4
5   void mm_method6(){}
6
7   public void trigger(){
8     SavingsAccount i1 = new SavingsAccount();
9     i1.mm_method6();
10  }
11
12 }
13
14
15 SubAccount class:
16 public class SubAccount {
17
18 }
```

Note that no trigger method is added to the `SubAccount`, since no methods were created for it.

Connection class:

```
1 package DataBase.package_p1;
2
3 public class Connection {
4
5   public void getConnection()
6   {
7   }
8
9   public void trigger()
10  {
11    Connection i1 = new Connection();
12    i1.getConnection();
13  }
14
```

```
15 }
```

Class2 class:

```
1  package GUI.Layer1;
2
3  public class Class2 {
4
5    public void method2(){}
6
7      public void trigger()
8      {
9        Class2 i1 = new Class2();
10       i1.method2();
11     }
12
13 }
```

Banking class:

```
1  package logic;
2
3  public class Banking {
4
5    public int[][] getGreenAccount(int i){
6      return new int[1][1];
7    }
8
9    public void trigger(){
10     Banking d1 = new Banking();
11     d1.getGreenAccount(0);
12   }
13
14 }
```

ClassCls2 class:

```
1  package logic.SubLogic;
2
3  public class ClassCls2 {
4
5    public void mm_method6(){}
6
7    public void trigger()
8    {
9      ClassCls2 i1 = new ClassCls2();
10     i1.mm_method6();
11   }
12 }
```

5.4.2 The `Helper` Class

We then create a `Helper` class to exercise the aspect using the `trigger()` method of each class. We only instantiate and access the classes that have trigger method defined in them, to trigger the aspect behavior

Helper class:

```
1  import DataBase.package_p1.Connection;
2  import GUI.Layer1.Class2;
3  import logic.Banking;
4  import logic.SubLogic.ClassCls2;
5
6
7  public class Helper {
8
9    public static void main(String args[]){
10
11     Account i1 = new Account();
12     i1.trigger();
13
14     AccountNumber i2 = new AccountNumber();
15     i2.trigger();
16
17     RunnableWithReturn i3 = new RunnableWithReturn();
18     i3.trigger();
19
20     SavingsAccount i4 = new SavingsAccount();
21     i4.trigger();
22
23     Connection i5 = new Connection();
24     i5.trigger();
25
26     Class2 i6 = new Class2();
27     i6.trigger();
28
29     Banking i7 = new Banking();
30     i7.trigger();
31
32     ClassCls2 i8 = new ClassCls2();
33     i8.trigger();
34   }
35 }
```

As you can see we have imported the appropriate classes to be able to create instances of the classes. And we use the first `public` constructor to create the instances. If there is no `public` constructor, we use the first `getInstance()` method.

Since the `trigger()` method we added to each class is not really part of the

original application code, we need to make sure it is transparent to the aspect, so no aspect behavior gets associated with it. We define 2 pointcuts, `excludeMyMethodsFromCall()`, and `excludeMyMethodsFromExecution()` that exclude `trigger()` methods from the call and execution pointcuts respectively. We add these pointcuts to the advice's point cuts which refer to methods names with a * wild card.

```
1 pointcut excludeMyMethodsFromCall():
2     !call (void *.trigger() );
3
4 pointcut excludeMyMethodsFromExecution():
5     !execution (void *.trigger() );
```

We also need to make sure that none of the pointcuts are applied to the `Helper` class since it is not part of the original application code. We do that by defining a pointcut `excludeMyHelperClass()` that excludes the `Helper` class, and we add it to each advice's pointcut in the aspect code.

```
1 pointcut excludeMyHelperClass():
2     !within(Helper);
```

So we rewrite the aspect code adding the `excludeMyHelperClass()` from each pointcut, and adding `excludeMyMethodsFromCall()` and `excludeMyMethodsFromExecution()` where appropriate. The following is the aspect rewritten:

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public aspect AccountAspect {
5
6    pointcut excludeMyMethodsFromCall():
7      !call (void *.trigger() );
8
9
10   pointcut excludeMyMethodsFromExecution():
11     !execution (void *.trigger() );
12
13
14   pointcut excludeMyHelperClass():
15     !within(Helper);
16
17
18   before():
19     preinitialization(private Account.new(int) throws Exception) &&
20       excludeMyHelperClass(){
```

```
21      System.out.println("Account␣preinit␣"+thisJoinPoint);
22    }
23
24
25    pointcut publicConstructor():
26      execution(public new()) && this(Account) && excludeMyHelperClass();
27
28    after():
29      publicConstructor() && excludeMyHelperClass(){
30      System.out.println("Public␣Constructor␣"+ thisJoinPoint);
31    }
32
33
34    pointcut creditAccount(Account account, float amount):
35      call( void credit(float)) && target(account) && args(amount) &&
36        within(Account) && excludeMyHelperClass();
37
38    before( Account account, float amount):
39      creditAccount(account, amount) && excludeMyHelperClass(){
40      System.out.println("Crediting␣" + amount +"␣to␣" + account );
41    }
42
43
44    before():
45      set( private float Account.balance ) && excludeMyHelperClass(){
46      System.out.println("Set␣Account.balance␣" + thisJoinPoint);
47    }
48
49
50    after():
51      get( !private * Account.*) && excludeMyHelperClass(){
52      System.out.println(thisJoinPoint +
53        "␣accessing␣a␣non␣private␣data␣member");
54    }
55
56
57    after() returning(float a1):
58      call(public * Account.getBalance(AccountNumber,int)) &&
59        excludeMyHelperClass(){
60      System.out.println("After␣Returning");
61    }
62
63
64    pointcut debitPointcut():
65      call(private void Account.debit(float) throws
66        InsufficientBalanceException);
67
68    after():
69      debitPointcut() && excludeMyHelperClass(){
70      System.out.println("Debit␣Pointcut");
71    }
72
73
74    pointcut methodWithincodeMethodPointcut():
75      call(public void AccountNumber.method1(int)) &&
76        withincode(private int Account.method1());
77
78    before():
```

```
79      methodWithincodeMethodPointcut() && excludeMyHelperClass(){
80      System.out.println("method␣withincode␣method␣pointcut:␣"
81        + thisJoinPoint);
82    }
83
84
85    pointcut constructorWithincodeMethodPointcut():
86      call(public SavingsAccount.new()) &&
87        withincode(private void Account.addSavingsAccount(int));
88
89    after():
90      constructorWithincodeMethodPointcut() &&
91        excludeMyHelperClass(){
92      System.out.println(
93        "constructor␣withincode␣method␣pointcut:␣" + thisJoinPoint);
94    }
95
96
97    pointcut startOfMethodName():
98      call(public void Account.new*());
99
100   before():
101     startOfMethodName() && excludeMyHelperClass(){
102     System.out.println("matches␣start␣of␣method␣name:␣"
103       + thisJoinPoint);
104   }
105
106
107   pointcut endOfMethodName():
108     call(private void Account.*bit(..));
109
110   before():
111     endOfMethodName() && excludeMyHelperClass(){
112     System.out.println("matches␣end␣of␣method␣name:␣"
113        + thisJoinPoint);
114   }
115
116
117   pointcut startAndEndOfMethodName():
118     execution(private void add*Account(*));
119
120   after():
121     startAndEndOfMethodName() && excludeMyHelperClass(){
122     System.out.println("matches␣start␣and␣end␣of␣method␣name:␣"
123        + thisJoinPoint);
124   }
125
126
127   pointcut anyReturnType():
128     call(public * Account.getBalance());
129
130   int around():
131     anyReturnType() && excludeMyHelperClass(){
132     System.out.println("Any␣return␣type:␣" + thisJoinPoint);
133     return proceed();
134   }
135
136
```

```
137   before():
138     call(public float Account.getBalance(*, ..)) &&
139        excludeMyHelperClass(){
140     System.out.println("* and ... arguments matches @: "
141        + thisJoinPoint);
142   }
143
144
145   after():
146     call(public int Account.*(int)) && excludeMyHelperClass(){
147     System.out.println("Any method name matches @: " + thisJoinPoint);
148   }
149
150
151   before():
152     call(public !final * Account.*()) && excludeMyHelperClass() &&
153        excludeMyMethodsFromCall(){
154     System.out.println("public non final matches @: "
155        + thisJoinPoint);
156   }
157
158
159   after():
160     execution(* *(..)) && within(SavingsAccount+) &&
161        excludeMyHelperClass() && excludeMyMethodsFromExecution(){
162     System.out.println("Any method SavingsAccount name matches @: "
163         + thisJoinPoint );
164   }
165
166
167   pointcut getGreenAccounts(int type):
168     call(int[][] logic.Banking.getGreenAccount(int)) && args(type);
169
170   after(int i):
171     getGreenAccounts(i) && excludeMyHelperClass() &&
172         excludeMyMethodsFromCall(){
173     System.out.println("packages: " + thisJoinPoint);
174   }
175
176
177   before():
178     call(void JComponent.repaint(..)) && excludeMyHelperClass(){
179     System.out.println("System defined:" + thisJoinPoint);
180   }
181
182
183   pointcut guiPackage():
184     call( public * GUI.Layer1.*.*() );
185
186   after():
187     guiPackage() && excludeMyHelperClass() &&
188         excludeMyMethodsFromCall(){
189     System.out.println("2 layer package " + thisJoinPoint);
190   }
191
192
193   declare parents: SubAccount extends Account;
194
```

```
195
196   after():
197     call(private int method1()) && within(Account || SavingsAccount)
198         && excludeMyHelperClass(){
199     System.out.println("Within␣either␣class" + thisJoinPoint);
200   }
201
202
203   after():
204     call( * GUI.*.Class2.*()) && excludeMyHelperClass()
205         && excludeMyMethodsFromCall(){
206     System.out.println("Any␣method␣in␣Class2,
207 ␣␣␣␣␣␣␣␣Class2␣belongs␣to␣a␣sub␣package␣of␣package␣GUI");
208   }
209
210
211   after():
212     execution( public * DataBase.*.Connection.getConnection() )
213         && excludeMyHelperClass(){
214     System.out.println("Any␣sub␣package␣in␣DataBase␣package");
215   }
216
217
218   before():
219     call( public void printStatement()) && !within(Account) &&
220         within(!SubAccount){
221     System.out.println("!within(Account)␣and␣within(!SubAccount)"
222         + thisJoinPoint);
223   }
224
225
226   after():
227     call(public * logic.SubLogic.*.*()) && excludeMyHelperClass()
228         && excludeMyMethodsFromCall(){
229     System.out.println("Create␣a␣new␣Class");
230   }
231
232
233   pointcut exceptionHandler():
234     handler(InsufficientBalanceException);
235
236   after():
237     exceptionHandler() && excludeMyHelperClass() {
238     System.out.println("Exception␣Handler␣matches␣@:␣"
239         + thisJoinPoint);
240   }
241
242   int Account.x = 0;
243   before():
244     set( int Account.x ) && excludeMyHelperClass(){
245     System.out.println("Set␣Account.x␣ITD␣" + thisJoinPoint);
246   }
247
248
249   void aspectMethod()
250   {
251     Account.print("My␣String");
252     RunnableWithReturn worker = new RunnableWithReturn();
```

```
253     try{
254       EventQueue.invokeAndWait(worker);
255     } catch (Exception e){}
256   }
257
258 }
```

5.5 Execution Output

The following will be the output when run after the aspect code and the constructed application code are woven together:

```
Set Account.x ITD set(int Account.x)
Public Constructor execution(Account())
Account preinit preinitialization(Account(int))
Set Account.x ITD set(int Account.x)
Set Account.x ITD set(int Account.x)
Public Constructor execution(Account())
Set Account.balance set(float Account.balance)
get(int Account.f1) accessing a non private data member
Crediting 0.0 to Account@32c41a
* and .. arguments matches @: call(float Account.getBalance(AccountNumber, int))
After Returning
method withincode method pointcut: call(void AccountNumber.method1(int))
Within either classcall(int Account.method1())
constructor withincode method pointcut: call(SavingsAccount())
matches start and end of method name: execution(void Account.addSavingsAccount(int))
matches start of method name: call(void Account.new_m3())
public non final matches @: call(void Account.new_m3())
Any return type: call(int Account.getBalance())
public non final matches @: call(int Account.getBalance())
Any method name matches @: call(int Account.mm_method4(int))
matches end of method name: call(void Account.debit(float))
Debit Pointcut
```

```
!within(Account) and within(!SubAccount)call(void AccountNumber.printStatement())
Any method SavingsAccount name matches @: execution(void SavingsAccount.mm_method5())
Any sub package in DataBase package
2 layer package call(void GUI.Layer1.Class2.method2())
Any method in Class2, Class2 belongs to a sub package of package GUI
packages: call(int[][] logic.Banking.getGreenAccount(int))
Create a new Class
```

CHAPTER VI

Related Work

Aspect-Oriented Programming is relatively a new topic and is a rich field for research. In this chapter we present what others have previously found that relates to our research.

Xie and Zhao [23] developed Aspectra, a framework for testing aspectual behavior that automates test inputs generation. As with our research, they also start with the aspects, and the developers construct the base classes, but they do not have a defined procedure in place on how to create the base classes. In addition, they use the byte code of the woven classes to create a wrapper class for each base class to invoke the calling join points. They had to do several runs, modifying the base classes to test the aspects. And in their paper they mention that a sophisticated base class construction tool is needed to help enhance the aspectual branch coverage testing. Our research defines how the base classes should be constructed.

Harman et al. [10] developed an approach for automating test data generation for AOP based on search-based optimization for hard-to-cover branches. Their approach produces test data for the base class and this data indirectly exercises the as-

pect, and their evolutionary tester generates test data for relevant parameters. They used domain reduction and program slicing, cutting off the unneeded parts of the program, which are not affected by the aspect behavior. In our research we take the opposite approach – that of creating the minimal set of application classes.

Zhou et al. [25] were also interested in testing aspects. Their testing approach consists of four steps. The first step is to test the classes by themselves, without the aspects in order to isolate and eliminate errors that are not aspect-related. The second step is to weave each aspect separately with the classes, and each woven application gets tested by itself to verify it is behaving as expected. The third step is to start weaving multiple aspects with the classes in an incremental fashion. The fourth step is to have all the aspects woven together with the classes to form the complete application, and test it. They define rules to select relevant test cases for the aspect-under-test. They are reusing test cases developed for the regular classes to test the aspect. New test cases are developed if the reused test cases do not cover the aspect under test.

J. Zhao [24] proposed a data-flow-based unit-testing approach for aspect-oriented programs. The research was concerned with testing the aspect, and the classes whose behavior are affected by the aspect code. They perform three levels of testing for each aspect or class: intra-module, inter-module, and intra-aspect or intra-class testing. Intra-module is for an individual module such as a piece of advice, a piece of introduction, and a method. Inter-module is for a public module along with other modules it calls in an aspect or class. Intra-aspect or intra-class is for modules that can be accessed outside the aspect or class, and can be invoked in any order by users of the aspect or class. Their approach uses control flow graphs to compute definition-use pairs to guide the selection of tests for the aspect or class. Their research does not cover inheritance in an aspect oriented program.

Interested in aspect behavior, Popovici et al. [16] developed a platform that allows for dynamic weaving called the PROSE (PROgrammable extenSions of sErvices) system. Aspects can be woven and unwoven at runtime. PROSE allows the creation of aspects using pure Java classes based on the PROSE library, which can be compiled using standard Java complier, and woven into the application at runtime. Aspect testing and validation are speeded up by the repeated weaving and unweaving of aspects.

Also interested in the impact of the aspect on the application behavior, Cavallaro and Monga [7] introduced an application prototype that performs change impact analysis on AspectJ programs. The tool can relate changes in an AspectJ program's source code to changes in the program behavior. They implemented their tool on top of the abc weaver.

Borger et al. [8], interested in runtime visibility and traceability of aspects, have implemented a debugger for AOP. This debugger supports the visibility of code abstractions and artifacts as the aspects, advices, pointcuts, aspect instances, and advice applications. The debugger supports the traceability of the advices and the join points that caused them on the stack. They implemented a breakpoint model on join points that allows the inspection of executed advices, executing advices, and future advices, corresponding to before, at, and after the breakpoint.

Another interesting research is the one done by Vidal et al. [22]. They were kind of doing the opposite of our research. Impressed by how well Aspect Oriented Programming preserves encapsulation and modularization of cross cutting concerns, they introduced a process to help developers refactor their Object Oriented applications into aspects. They developed a tool approach that uses aspect mining with rule base engine to apply refactorings.

Avgustinov et al. [5] were concerned about the safety of the pointcuts. They

introduced rigorous semantics for AspectJ pointcut language. Their approach rewrites the pointcuts in Datalog queries, a prolog-like language.

Krishnamurthi et al. developed a technique to verify the aspect advice modularly [14]. They assumed that the pointcut is fixed, and the advice is the one that changes by the developer. This technique eliminates unnecessary analysis of the entire system, every time the developer makes a change to the advice. Their verification technique takes as input the programs with the pointcuts without the advices, and a set of properties that the advice must not violate. They use finite state machine to represent the aspect-oriented program, and apply model checking against them. Their research also considered optimization of multiple pointcuts designators. As described before, in our approach we do consider all parts of the aspect including the advice, since we depend on the provided aspect code to find the clues about the original application.

6.1 Implementation

During our research, we explored options for how we would implement a tool to do the reverse engineering on the aspect code and construct the application code. AspectJ Front [2, 6] was one of the options we came across. We found out that this project is not active any more, and we needed a more flexible framework to work with, to be able to change it and build on it. So we started exploring other options.

We looked at AspectBench Compiler (abc) [3, 4], a compiler framework. AspectBench is designed with extensibility in mind, which makes building on new features easy. It is built in java, and it implements full AspectJ language. It allows extensions in: syntax, type checking, code generations, data and control flow analyses. We talk more about it in future work section.

Another option that is worth evaluating, is to consider building our tool on top

of the java eclipse aspectj project [1]. It is an open source project. Eclipse provides a java IDE framework, and it has an aspectJ plug-in for aspect oriented programming.

6.2 Non Aspect Oriented

Finding the application elements from the join points that the aspect behavior would apply to resembles implementing frameworks where the methods are left virtual or abstract [17, 18, 21]. The application which is built using the framework is the one that defines the behavior of these abstract methods. In this case the framework defines the virtual method (the template method) and the application defines the behavior of the hook method.

Another way for decoupling cross-cutting concerns is to use software containers. Cross-cutting concerns as persistence, security, transaction management, and fault masking are implemented as container services. Sridhar and Hallstrom [19] present a formal model that allows the developers to identify how the behavior of components deployed in a container are modified by the container.

CHAPTER VII

Conclusions and Future Work

7.1 Conclusion

In this research we have introduced a reverse engineering approach that allows us to examine aspect behavior separately from the application and before weaving into the application is done. We have also defined the transformation rules needed to reverse engineer the application from the aspect code. In this thesis we have presented an example that illustrates step by step how our approach and the transformation rules are applied to construct the application class model.

Our approach takes the aspect code as its input, and finds clues of the elements the aspect(s) expect(s) to see. Using the transformation rules, we build the constructed application classes. Our approach includes several passes to identify the packages, classes, fields, methods, and constructors of the application. The constructed application is the minimal subset of the original application; it does not have any behavior in itself. The constructed application classes are basically empty. The aspect and the new constructed application are then woven together. The output

behavior of the woven application is exclusively pure aspect behavior.

7.2 Future Work

7.2.1 Tool Implementation

The immediate and most pressing extension of this work is the implementation of a tool that reverse engineers aspect code into the constructed application code. As mentioned before, we looked into ways for the tool implementation, and we feel implementing on top of the AspectBench Compiler (abc) compiler is suitable for our purpose. The AspectBench Compiler is designed with extensibility in mind, for easy experimentation with new language features and implementation techniques for AspectJ. It allows extensions in: syntax, type checking, code generations, data and control flow analyses. It allows easy add-ons, keeping new extensions and base code disentangled.

AspectBench [3, 4] compiler is built in java and implements the full AspectJ language. Its front end is built on Polyglot, which provide syntax flexibility and type checking. Its back end is built on Soot, for modular code generation, analyses and code weaving. abc is freely available under the GNU LGPL.

Another option that is worth evaluation, is to consider building our tool on top of the java eclipse aspectj project [1]. It is a open source project. eclipse is an provides a java IDE framework, and it has a aspectJ plug-in for aspect oriented programming.

7.2.2 Test case generation

One of the applications of our approach is test case generation. A number of other researchers have been working on testing of aspect-oriented programs in the recent years [10]. The approach usually is that the application classes are woven

along with the aspects, and the whole application is analyzed to determine test cases. In this manner, test cases can be computed that completely cover and exercise the various behaviors exercised by the application. Some researchers have also worked on isolating the behavior encapsulated in the classes from that encapsulated in the aspects [12]. Our approach of constructing structural harnesses for aspects allows the use of the same test generation techniques, while completely isolating and testing only the behavior exhibited by the aspects.

7.2.3 Design pattern mining

Another application of our approach is design pattern mining. Hannemann and Kiczales [9] have shown the use of aspects in refactoring the design of a system to use one or more design patterns. To do this, the design pattern is encoded as a set of class relationships in a target application. Since our reverse engineering process results in a structural subset of the target application's class model, generating a harness for the aspect(s) that implement(s) a certain design pattern, and then comparing this harness with the complete class model of an application can allow us to identify the existence of design patterns or idioms in legacy Object Oriented applications.

7.2.4 Design model validation

We can also use our approach in design model validation. The harness classes that we can reverse engineer from the aspect(s) are structurally a subset of the target application. As such, dependencies and inheritance relationships between classes in the target application can be inferred insofar as they are captured in the aspects. Based on these inferences, the design model of the target application can be validated against specifications that require (or disallow) certain relationships among classes.

BIBLIOGRAPHY

[1] Aspectj at eclipse.org. http://www.eclipse.org/aspectj.

[2] Aspectjfront website. http://strategoxt.org/Stratego/AspectJFront, 2006.

[3] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Building the abc aspectj compiler with polyglot and soot. 2004 abc-2004-4, aspectbench.org.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[5] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in aspectj. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 11–23, New York, NY, USA, 2007. ACM.

[6] M. Bravenboer, E. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for aspectj. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 209–228, New York, NY, USA, 2006. ACM.

[7] L. Cavallaro and M. Monga. Unweaving the impact of aspect changes in aspectj. In *FOAL '09: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 13–18, New York, NY, USA, 2009. ACM.

[8] W. De Borger, B. Lagaisse, and W. Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 173–184, New York, NY, USA, 2009. ACM.

[9] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.

[10] M. Harman, F. Islam, T. Xie, and S. Wappler. Automated test data generation for aspect-oriented programs. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 185–196, New York, NY, USA, 2009. ACM.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In S. M. Mehmet Aksit, editor, *11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[12] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM.

[13] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.

[14] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM.

[15] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[16] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM.

[17] N. Soundarajan and S. Fridella. Understanding oo frameworks and applications: an incremental approach. *Informatica (Slovenia)*, 25(3), 2001.

[18] N. Soundarajan and B. Tyler. Testing polymorphic behavior. *Journal of Object Technology*, 1(3):173–188, 2002.

[19] N. Sridhar and J. O. Hallstrom. A behavioral model for software containers. In *FASE*, pages 139–154, 2006.

[20] Sun Microsystems. The java reflection API. http://java.sun.com/docs/books/tutorial/reflect/index.html, 2010.

[21] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In *FATES*, pages 1–14, 2003.

[22] S. Vidal, E. S. Abait, C. Marcos, S. Casas, and J. A. Díaz Pace. Aspect mining meets rule-based refactoring. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 23–27, New York, NY, USA, 2009. ACM.

[23] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 190–201, New York, NY, USA, 2006. ACM.

[24] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 188, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Y. Zhou, H. Ziv, and D. J. Richardson. Towards a practical approach to test aspect-oriented software. In *SOQUA/TECOS*, pages 1–16, 2004.