An Investigation of Modular Dependencies in Aspects, Features and Classes

By

Shoushen Yang

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Master of Science

In

Computer Science

by

_____

May 2007

APPROVED:

_____

Professor George T. Heineman, Thesis Advisor

_____

Professor Gary Pollice, Thesis Reader

_____

Professor Michael Gennert, Head of Department

ABSTRACT

The essence of software design is to construct well-defined, encapsulated modules that are composed together to build the desired software application. There are several design paradigms in use today, including traditional Object-Oriented Programming (**OOP**), Feature-Oriented Programming (**FOP**), Aspect-Oriented Programming (**AOP**) and Instance-Oriented Programming (**IOP**). **FOP** studies the modularity of features in product lines, where a feature is an increment in program functionality. **AOP** aims to separate and modularize aspects when an aspect is a crosscutting concern. **IOP**, as an extension to **FOP**, makes the layers work like object factories. While each is good at solving different types of problems, they are closely related. The composition of modules is complicated because modules have (often hidden) dependencies on other modules. This thesis aims to better understand the way *dependencies* are managed by each approach. We focus on the dependencies caused by the inheritance relationship in **OOD**. We also focus on the precedence issue in **AOP** and **FOP**, that is, how designers are able to specify the order by which modules are composed together. Different precedence means different semantics, but the current tools can not guarantee the correct precedence is adopted. We first provide a way to help the designer better manage the dependencies in **OOD**, then we solve the precedence issue separately for **AOP** and **FOP** by providing a similar way to manage the dependencies in **AOP** and **FOP**. Based on this, we show that a unified model is possible to help manage the dependencies by using source code annotations to specify the designer intent. We evaluate our technique with use cases.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor George Heineman, for his great help on my study. Without his guidance and support, I do not think I can finish my thesis. Professor Heineman is a great advisor and programmer. He is the one I should learn from now and in my future career. Great thanks are also to my thesis reader, Professor Gary Pollice.

I also want to thank WPI Computer Science Department, which gave me the chance to study and work to get my Master Degree. This is a very important milestone in my career. I hope I can do something to WPI as a return in the future. Besides, I would like to thank all the faculty members who have helped me in the past two years.

1 Introduction

While there are numerous approaches to software design, they are based on the common idea of creating a design from modular building blocks. Over time, a design is expanded or changed by adding new modular building blocks to the design. In this thesis, we focus on the question of adding a new module M_{n+1} to a design. This can be represented as the composition of a set of design modules:

$$D = M_1 \bullet M_2 \bullet M_3 \bullet \dots \dots \bullet M_n \bullet M_{n+1}$$

where ● is some composition operator. Assuming the compositions occurs from left to right, then

$$D = ((((M_1 \bullet M_2) \bullet M_3) \bullet \dots \dots \bullet M_n) \bullet M_{n+1})$$

Given this model theory, we need to understand:

1. what is possible to be a modular unit, and
2. what is possible for the composition operator ●.

This design question is more limited than the issue of general design because the modules M_1, M_2, … , M_n are not changed when we add a new module M_{n+1}. In this thesis, we investigate four design approaches: Object-Oriented Programming (**OOP**), Feature-Oriented Programming (**FOP**), Aspect-Oriented Programming (**AOP**) and Instance-Oriented Programming (**IOP**).

In **OOP**, the modular units are simply classes, which may form a package as one modular unit. The other three approaches can all contain classes as units; however, in **AOP** the modular units may be aspects; in **FOP**, they may be features, composed with a set of classes or refined classes; while in **IOP**, they may be a set of instances.

Regarding the linear composition shown above, it may happen that a modular unit is dependent on some other modular unit. These dependencies are complicated because software composition has an inherent order. Consider two functions *f* and *g* with the following definitions:

$$f(x) = x + 5;$$

$$g(x) = 2x;$$

The composition between *f* and *g* has different result if we use different orders:

$$f \bullet g\ (x) = 2x + 5;$$
$$g \bullet f\ (x) = 2(x + 5) = 2x + 10;$$

Composing modular units is similar to composing functions. "$M_1 \bullet M_2$" may be different from "$M_2 \bullet M_1$".

There is a critical problem related to dependencies between modular units, namely how the precedence of the units impacts the semantic meaning or behavior of the software. There is some work related to this issue [18], but no general framework on guaranteeing correct precedence between the modular units. For example, Stoerzer et al in [18] demonstrated the precedence issue, analyzed the interference between **AOP** advice, provided a solution to detect the conflict between aspects, and came up with an implementation to tell the programmer what aspects the explicit precedence should be declared between. However, from the perspective of the designer [18] still could not tell when if a wrong explicit precedence was declared.

For **OOP**, the composition is complex due to inheritance, association and aggregation relationships between the classes. What we most care about in this thesis is the composition caused by inheritance, because inheritance can make a newly-added modular unit inherit code from an existing unit, and this leads to the question of how the old code is inherited. We know that in Java the overriden method uses `super` to directly access code found in a superclass, but the use of `super` may have many possibilities. At this point, **OOP** is similar to **FOP**, which we now explain.

For **FOP** or **IOP**, the composition is linear and straightforward; however the composition is more complex than it appears. In **FOP**, take the AHEAD suite [2] for example; while layers are composed linearly, the order of layers within the equation barely captures the relationship between them. Inspecting the code of the layers more closely, one may find that the superficial linear composition does not describe the semantics of the final generated code. The simplest example is that in **FOP**, if the programmer forgets to call `Super()`, the relationship will not exist between this layer and its parent layers; in addition, the refined layer can call `Super()` at the beginning of the refined method, or at the end of the code. We can treat this situation as caused by the precedence issue between the layers and the `Super()` invocation positions. In **FOP** the current strategy to guarantee the sequence of generated code is not enough. When considering **FOP**, we consider **IOP** at the same time by the ACDK project at WPI [3], because

IOP is based on the idea of **FOP** and provides the extension of the concept of instance.

The composition in **AOP** is more complex due to the flexible woven strategy. A piece of advice can be embedded into any number of classes at the same time, by using wildcards in the *joinpoints*. When more and more aspects and classes are woven together, the final program may be unpredictable. And as many papers have pointed out that, different conflicts may exist between the different modular units [22] [23] [24]. That is, the composition may encounter some conflict problems. The precedence between the aspects during the composition is one of the possible reasons which cause the conflicts, and the issue is just what we will focus in this thesis. We consider the AspectJ framework [14] for **AOP**. AspectJ allows the programmers to specify the composition precedence between aspects by declaring precedence and the precedence between advice and the method in objects by declaring the advice to be *before*, *after* or *around* the joinpoint. However, similar to AHEAD, this is not enough as well. There is no way to guarantee a correct sequence is applied during the composition process. If some incorrect precedence is used, the programmer has to wait till the runtime to find it.

In this thesis, we will solve the precedence problem for **AOP**, **FOP** and **IOP** separately, and define a unified model. We first analyze dependencies between the units by analyzing the data flow and control flow interference between aspects, features or classes. We then analyze which kinds of dependencies are useful to precedence by categorizing the dependencies. We finally investigate how to detect and solve the potential wrong semantics caused by wrong precedence, and come up with a unified model to resolve the precedence issue by using source code annotations. We test our solution with enough use cases.

1.1 Motivation

To the best of our knowledge, there is no general solution to guarantee the correct precedence of composition in **IOP**, **FOP** and **AOP**. Moreover, because of the similarity of **AOP**, **IOP** and **FOP**, the solutions in each methodology should be similar as well. Actually for **FOP**, there is only discussion on the precedence between layers in an equation which is insufficient as we demonstrate. In the next subsections, we will use examples to demonstrate why this issue is critical in **AOP**, **IOP** and **FOP**. Due to the similarity between **IOP** and **FOP**, we only show the example for **FOP**.

1.1.1 An AOP Example

For **AOP** research, the aspect interference problem is a known issue [18]. The advice precedence problem is one example, where different precedence of aspects can cause different semantics. Here is a simplified TeleCom example taken from the AspectJ distribution [14].

We will show step by step how the development process looks like for the simplified Tele-system and finally come up with the precedence problem between the aspects. Please note in this example we are not considering the multi-thread condition, which means we assume in the system there is only one phone call being made at a time.

Let's suppose the initial program for the system is like Figure 1. We only have the `Connection` class to manage the connection and there is no requirement to calculate how much the customer should pay for the calling (Assume at the beginning any call is free).

```
public class Connection{
   void start(){
    System.out.println("Starting the call.");
  }
  void stop(){
    System.out.println("Stopping the call.");
  }
  public static void main(String[] args) throws InterruptedException{
       Connection conn = new Connection();
       conn.start();
       //the customer is in the call
      System.out.println("Calling...");
       Thread.sleep(1000);
       conn.stop();
     }
}
```

The output:

```
Starting the call.
Calling...
Stopping the call.
```

Figure 1. Base Class – Connection

Now we have the new requirement which asks to calculate how long every phone call lasts so that we can calculate how much every customer should pay. We plan to adopt **AOP** for the requirement change. We can use one aspect called *Timing* to calculate the time and one aspect

called *Billing* to calculate the bill. We know after a connection is stopped, the *Timing* aspect should calculate the time and the *Billing* aspect should calculate the bill. We have two programmers to work on this and after a discussion they decide to add a helper class called `Timer` as Figure 2 shows. The `Timer` class provides the ability to store the starting time and the ending time, and finally determine how long a phone call is. The *Timing* aspect needs to keep the starting and ending time by calling the `Timer` class. The *Billing* aspect needs to call `Timer` to get the time then calculate the bill after one connection is stopped. Then two of the programmers begin to work on each of the aspects.

The two aspects are finished as Figure 3 shows. The programmer who is responsible for *Timing* has tested the aspect and is confident in it. The output is shown in Figure 4. At the same time the programmer for *Billing* also claims that he/she has done enough testing and the aspect works well.

```
public class Timer{
  public static long start, stop;
  public static void start() {
   start = System.currentTimeMillis();
   System.out.println("Starting time:" + startTime);
   stop = start;
  }
  public static void stop() {
   stop = System.currentTimeMillis();
   System.out.println("Stopping time:" + stopTime);
  }
  public static long getTime() {
   return stop - start;
  }
}
```

Figure 2. Helper Class – `Timer`

```
public aspect Timing{
   pointcut start (): call(void
Connection.start());
   before(): start () {
   Timer.start();
   }
   pointcut end (): call(void
    Connection.stop());
   after(): end () {
   Timer.stop();
   }
}
```

```
Public aspect Billing{
   private long rate = 10;
   pointcut end (): call(void
   Connection.stop());
   after():end () {
     long time = Timer.getTime();
    long cost = rate * time;
    System.out.println("The cost
    is: " + cost);
  }
}
```

Figure 3. Aspects – *Timing* and *Billing*

```
The output:
Starting time:1169736011125
Starting the call.
Calling...
Stopping the call.
Stopping time:1169736012156
```

Figure 4. Output when weaving only aspect *Timing*

Then they begin to integrate the two aspects with the old code. However after the integration they find the output is not as expected, as shown in Figure 5.

```
The output:
Starting time:1169751255406
Starting the call.
Calling...
Stopping the call.
The cost is: 0
Stopping time:1169751256406
```

Figure 5. Output when weaving two aspects – *Timing* and *Billing*

From the output we can see the cost is 0. So they begin to investigate in and they determine that the bill is calculated before the time is calculated. And the reason is that a random precedence is used when no explicit declaration on this is made.

After they realize the reason they add a new aspect to specify the precedence between the two aspects, as shown in Figure 6.

```
public aspect Precedence{
    declare precedence: Timing, Billing;
}
```

Figure 6. Wrong aspect precedence

Then they weave all of them and run the program again. Unfortunately, they still get the same wrong output. The reason is that an after advice is used on the pointcut, so the preceded aspect is actually executed afterwards. Then they figure out the reason and change the order of the two aspects in the declare statement, as Figure 7 shows.

```
public aspect Precedence{
    declare precedence: Billing, Timing;
}
```

Figure 7. Correct aspect precedence

After they weave the three aspects together with the other two classes, they find that finally the problem is solved. Figure 8 shows the correct output.

```
The output:
Starting time:1169751279421
Starting the call.
Calling...
Stopping the call.
Stopping time:1169751280421
The cost is: 10000
```

Figure 8. Output when specifying the correct precedence

From this example, we can see the precedence issue may cause challenge, and the current strategy can not guarantee a correct semantics. The verification can only be done during runtime, and there is no efficient way to detect this kind of problem. We propose that one can detect invalid semantics during compile time.

1.1.2 An FOP Example

FOP provides no standard way to identify the precedence between features. The lack of similar discussion in **FOP** is the primary motivation for the proposal work. We argue that it is not good enough to only depend on the implicit precedence in an equation:

$$D = M_1 \bullet M_2 \bullet M_3 \bullet \ldots \ldots \bullet M_n \bullet M_{n+1}$$

We use the simplified TeleCom example from the prior section to demonstrate **FOP**. As in **AOP**, we will also show step by step how the development process looks like and finally come up with the precedence problem between the layers. Once again, assume in the system there is only one phone call at a time.

```
public class Connection{
    void start(){
        System.out.println("Starting the call.");
    }
    void stop()   {
        System.out.println("Stopping the call.");
    }
    public static void main(String[] args) throws InterruptedException
    {
        Connection conn = new Connection();
        conn.start();
         //the customer is in the call
        System.out.println("Calling...");
        Thread.sleep(1000);
        conn.stop();
    }
}
```

The output:

```
$ ./TO_RUN
Starting the call.
Calling...
Stopping the call.
```

Figure 9. Base Layer (Layer 0) – `Connection`

Let's suppose the initial program for the system is like Figure 9, which is the base layer (layer 0) and actually exactly the same as base program in **AOP**. We only have the `Connection` class to manage the connection and there is no requirement to calculate how much the customer should pay for the calling (Assume at the beginning any call is free). And of course the output is also the same in **AOP**.

Now we have the new requirement (again, same as in **AOP**) which asks to calculate how long every phone call lasts so that we can calculate how much every customer should pay. We plan to adopt **FOP** for the requirement change because we know it is also reasonable to use **FOP** for this example. We can have a layer called *Timing* to calculate the time and a layer called *Billing* to calculate the bill. We can also add a helper class called `Timer` as Figure 10 shows (same as in **AOP**). The `Timer` class provides the ability to store the starting time and the ending time, and finally provide how long a phone call is. We put `Timer` and *Timing* layer to calculate the time. We know after a connection is stopped, the *Timing* layer should calculate the time and the *Billing* layer should calculate the bill. So apparently *Timing* should be before *Billing*.

The two layers are finished as Figure 11 shows. By only applying *Timing* we can get the output as

shown in Figure 12. By applying *Billing* together with *Timing* we get the output as Figure 13 shows. Figure 14 shows the architecture in AHEAD.

```
public class Timer{
    public static long start, stop;
    public static void start() {
        start = System.currentTimeMillis();
        System.out.println("Starting time:" + start);
        stop = start;
    }
    public static void stop() {
        stop = System.currentTimeMillis();
        System.out.println("Stopping time:" + stop);
    }
    public static long getTime() {
        return stop - start;
    }
}
```

Figure 10. Helper Class – `Timer`

However if we use the wrong precedence between the *Timing* layer and *Billing* layer, such as composing *Billing* before *Timing* (this may happen by mistake), we will get the wrong output as Figure 15 shows.

Even if we can guarantee the layers' precedence between each other, there are still potential problems. In the *Billing* layer in Figure 11, a `stop()` call to its `Super()` method (i.e., the layer which *Timing* refines) must be invoked before the other code. But what will happen if the programmer forgets to call it, or even put the wrong order with the other code? In both cases, we will get the wrong semantics. For example if we delete the `stop()` call to its `Super()` method, we will get the output as Figure 16 shows. And if we invoke the `Super()` at the end of the `stop()` method, we will get the wrong output as Figure 17 shows.

```
refines class Connection{
    void start(){
        Timer.start();
        Super().start();
    }
    void stop(){
        Super().stop();
        Timer.stop();
    }
}
```

```
refines class Connection{
    private long rate = 10;
    void stop(){
        Super().stop();
        long time = Timer.getTime();
        long cost = rate * time;
        System.out.println("The cost
is: " + cost);
    }
}
```

Figure 11. *Timing* (left) and *Billing* (right)

The output:

```
$ ./TO_RUN
Starting time:1170337275078
Starting the call.
Calling...
Stopping the call.
Stopping time:1170337276125
```

Figure 12. Output when weaving only *Timing*

The output:

```
$ ./TO_RUN
Starting time:1170337430640
Starting the call.
Calling...
Stopping the call.
Stopping time:1170337431687
The cost is: 10470
```

Figure 13. Output when weaving two layers – *Timing* and *Billing*

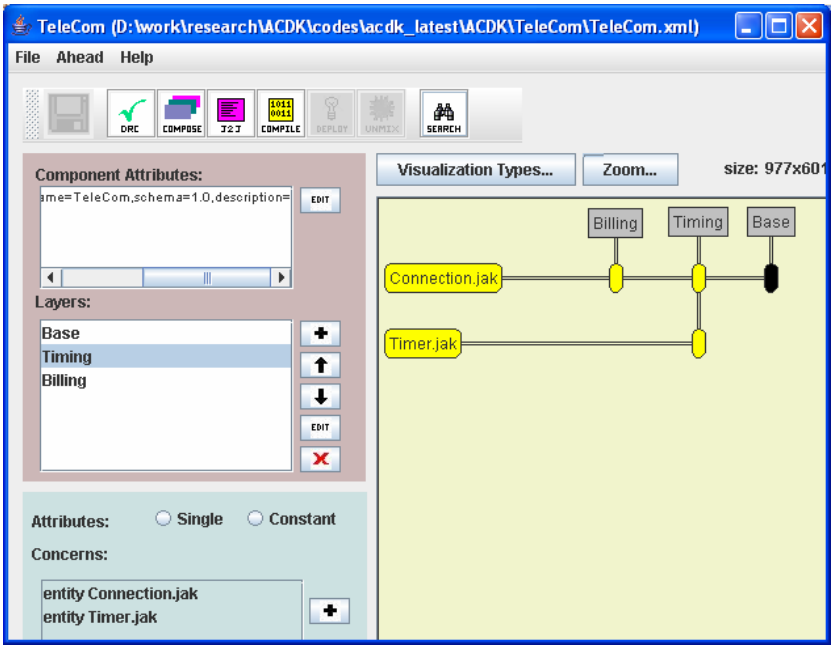

Figure 14. The whole architecture in AHEAD

The output:

```
$ ./TO_RUN
Starting time:1170337880468
Starting the call.
Calling...
Stopping the call.
The cost is: 0
Stopping time:1170337881515
```

Figure 15. Output when *Timing* is after *Billing*

The output:

```
$ ./TO_RUN
Starting the call.
Starting time:1170339024828
Calling...
The cost is: 0
```

Figure 16. Output when forgetting the call to `Super()` in *Billing*

The output:

```
$ ./TO_RUN
Starting the call.
Starting time:1170339426593
Calling...
The cost is: 0
Stopping the call.
Stopping time:1170339427640
```

Figure 17. Output when `Super()` is called after the other code in *Billing*

From the example, we can see, in AHEAD, the dependency is maintained by the `refines` and `Super()` keywords between layers together with the sequence of the layers. The equation file only provides the high-level structure of the system composition and there are numerous implementations that could alter, often greatly, the semantic understanding of the resulting system, by calling `Super()` at different places, or if the programmer misses the `Super()` invocation just through carelessness. There is no way to specify or capture this missing invocation. Thus it's not guaranteed when we want to do conservative weaving, which requires a `Super()` call must happen exactly once [10]. We need to not only guarantee the correctness of high-level structure, but also guarantee the correct invocation of the super call.

1.2 Related work

Comparison and contrast have been deeply discussed between **AOP** and **FOP**. For example [4] shows when to use features and when to use aspects by case study. At the same time, many researchers are working on unifying the different paradigms. [8] tries to unify **AOP** and traditional **OOD**. [17] uses Aspectual Mixin Layers to bring concert between **FOP** and **AOP**.

About the interference between features or aspects, [10] addresses the question of semantic reasoning about multiple weavings, by considering semantic refinement for **FOP** where components are built from features and weavings. [18] presents an interference analysis between the aspects and provides a solution to detect and solve the conflicts, but it can not guarantee a correct precedence is used. [22] aims to detect the semantics conflict between crosscutting concerns in **AOP**. In [24], a tool named Alpheus is given, which allows users to specify aspects dependencies and to specify conflicts and resolution rules. In [5], a framework is proposed to support conflict detection and resolution, which is more powerful than the *declare precedence* construct of AspectJ. [26] proposes a combined pointer and effect analysis to classify aspects by their effects on the base system. [27] addresses the increased complexity of AspectJ, and also

propose one solution to solve the precedence issue. [37] investigates into the incompatible and inconsistent interactions when multiple aspect languages are used when implementing a system. [37] also illustrates how to solve the adverse interactions.

ACDK is the ongoing research being conducted in WPI [3] to implement the idea of **IOP**. For the precedence issue, ACDK extends the AHEAD suite [2] to allow the users to specify concerns between layers. ACDK also provides an easy-to-use UI to view the concerns.

2 Background

2.1 Object-Oriented Design (OOD)

The idea of object-oriented (OO) originated in the 1960s; however it was not commonly used in mainstream software application development until the 1990s. Today most programming languages support Object-Oriented Programming. Object-oriented design (**OOD**) is concerned with designing a model by separating the requirements as different objects.

In the methodology of OO, the main composed element is called a *class*, which contains *constructors*, *methods* and *fields*. A class is generalized from objects which have the same characteristics in dynamic, run-time form. Once a class is defined, objects can be instantiated from it. Fields (also called attributes or properties) describe the information defined by a class and therefore maintained by an object. A method defines executable behavior for a class. Methods can be used to update the *state* of the object, where state is defined to be the set of the values of an object's fields.

A subclass can *extend* a class to create a new class. Subclasses can have new fields to describe the new characteristics of the subclasses and new methods that define their new behavior. Also, a subclass can change the methods that it inherited from its superclass (also known as the parent class) – such methods in the subclass are called *overriden* methods. The mechanism of inheritance introduces two concerns: how is the the constructor in the subclass defined, and how do overriden methods in the subclass alter the behavior of the method as defined in the parent class.

Constructors in a subclass must follow strict constraints. If the parent class uses a default constructor (i.e., a no-argument constructor) then the subclass can continue to use the default constructor or create its own constructor without any constraint. However, if the parent class has a constructor which is not the default one (i.e., there are typed parameters), the subclass must explicitly define a constructor, and in the constructor, a `super()` call to one of the constructors in the parent class must be explicitly invokes as the first statement of the constructor in the subclass. In addition, the `super()` constructor may only be invoked once. These constraints are

understandable since constructors are used to initialize objects, and a subclass must rely on its superclass to properly initialize the object from its perspective before the subclass can initialize the object. If the superclass were not initialized first, some inherited fields could be used without definition. All in all, constructors are special and strict constraints are necessary to guarantee that a correct object can be created from the class.

Regarding overriden methods, there are few constraints on how (or when) to call the `super` method. The subclass may never call the `super` method, or it could call the `super` method multiple times. We believe this lack of constraints will cause problems. For example, a parent class might expect that a `super` call is invoked in subclasses, but if it is not invocated (this is possibly done purposely, also maybe through carelessness), the compiler is unable to detect an error, or even flag a warning. A potential defect may arise. Besides, the location of the `super` invocation may generate different semantic code. We have conducted a deep analysis on this topic in Section 3.1 .

Note that coupling and cohesion are the way to evaluate the dependencies in traditional Object-Oriented Software Engineering (OOSE) and much research has been conducted on the topic (see [20] for a survey). However in **OOD** there is no way to explicitly capture dependencies between classes, which lead to fragile code. We try to capture the dependencies caused by inheritance to avoid the fragile code. More information on the **OOD** paradigm can be found easily (for example, [28] gives a good explanation).

2.2 Aspect-Oriented Programming (AOP)

The essence of software design is to construct well-defined, encapsulated modules that are composed together to build the desired software application. However, designers still have difficulty fully dividing a problem as a completely modular and encapsulated model. Although breaking a problem into objects makes sense, some pieces of functionality must be across objects. Aspect-Oriented Programming (**AOP**) is one of the solutions to the problem.

AOP is just a concept, so it is not restricted to a specific programming language. There have been different frameworks of **AOP** for the extensions of different programming languages, for example, the tools that support **AOP** for Java include: AspectJ, AspectWerkz, JBoss AOP etc, the tools for

C++ include: AspectC++ and FeatureC++, and tools for Ruby include AspectR. In this thesis, we will use AspectJ as the framework for the explanation of the concepts and examples. A small example would be provided to explain how to use **AOP** for a project. Some interesting stuff related to AspectJ would be introduced briefly. But before that, we will first introduce some concepts in **AOP**.

2.2.1 Basic Concepts

Aspect: An aspect is a modular unit of crosscutting implementation [36]. It encapsulates behaviors that can be injected into multiple classes as reusable modules. A typical implementation process with **AOP** is, we first implement our project, and then we do separately with crosscutting concerns in the code by implementing aspects. Finally, both the code and aspects are woven into a final executable form with an aspect weaver.

Crosscutting concerns: Though many classes in an OO model perform a single, specific function, they often share some common requirements with other classes. For instance, we may want to add logging to classes whenever a thread enters or exits a method. With the traditional OO paradigm, we have to have corresponding code in every class. Thus, even though the primary functionality of each class is very different, the code for secondary functionality is sometimes identical. Those pieces of secondary functionality are called crosscutting concerns.

Join Points: A join point means to define a point in the program flow, where advice can be woven into the code of an application. For example, a join point can be defined on the following actions:

- ✓ Method call, Method execution
- ✓ Constructor call, Constructor execution
- ✓ Object pre-initialization, Object initialization
- ✓ Field reference, Field set
- ✓ Handler execution
- ✓ Advice execution

An example of a join point for a method call would be like:

```
call(void Connection.stop())
```

The join point means the point when the `stop()` method in the Connection class is called.

For one statement of a join point, you can use wildcards to represent many join points, or plus (+) to represent the subclasses, for example:
`call(void *.stop())` means the points when the `stop()` method in any class is called.

Point-cut: A pointcut is a set of join points. This is the term given to the point of execution in the application at which crosscutting concern needs to be applied. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed. An example to define a pointcut is:

```
pointcut myPointCut(): call(void Connection.stop())||
                       call(void Connection.start());
```

This pointcut specifies a pointcut when the `stop()` or `start()` method is called in the Connection class.

Advice: This is the additional code that you want to apply to your existing application. The advice may have the following types:

- ✓ before
- ✓ around
- ✓ after
  - ■ after
  - ■ after returning
  - ■ after throwing

An example to a piece of advice is like:

```
after():myPointCut () {
        System.out.println("This is my advice");
    }
```

This advice means to add additional code after the previous pointcut is reached.

2.2.2 A Small Example

Suppose you are working on a banking system. You know in every operation of the account, the authentication must be performed to make sure that the user has the right. In traditional programming, you may include the authentication checking in every operation. But this way, a lot of redundant code would be generated (you may just use "copy + paste" to work on this, but this is prone to errors). If we use **AOP** to implement this, we can take the authentication as the crosscutting concern, and use an aspect to implement it, then weave it into the other code.

The code for the operations may be as Figure 18 shows (of course there may be multiple classes.).

```
public class Account{
    public void openAccount(){}
    public void closeAccount() {}
    public void transfer(){}
    ……
}
```

Figure 18. `Account` class

The aspect for this is possible to be as Figure 19 shows.

```
public aspect Authentication {
    pointcut authentication(): call(void *.*());

    void around(): authentication(){
        // do the authentication
        if (isCorrect(authentication))
            proceed();
        else
            throw new RuntimeException("Wrong authentication!");
    }
}
```

Figure 19. *Authentication* aspect

In this example, we defined the aspect to use an around advice to execute the authentication. We stated that if the authentication is passed, the exact account operation can be continued, but if the authentication is not passed, an exception would be thrown. The `proceed()` method will continue the execution of the method specified in the join points.

2.2.3 Advice ordering

You might be wondering, if there are multiple aspects or multiple sets of advice in one aspect, what is the order to apply them on the same join point? The answer is that the precedence can be declared between the aspects, and some default precedence is used inside one single aspect.

For advice within a single aspect, precedence among before and around advice follows a simple rule: a piece of advice declared earlier in the source file has higher precedence over a piece of advice defined later. However, after advice behaves differently: if two pieces of advice defined in the same aspect want to execute at the same join point, and one or both of them are after advice, the advice defined later in the source file has higher precedence over the one defined earlier [14].

For the precedence between aspects, the programmer can explicitly declare the precedence between them, for example, if there are two aspects A and B, the programmer can declare like this:

```
declare precedence: A, B;
```

And the advice in the higher aspects has higher precedence. Now, suppose we have the basic class as following as Figure 20 shows.

```
public class C {
    public void m()  {
        System.out.println("This is the basic method.");
    }
    public static void main(String[] args) {
        new C().m();
    }
}
```

Figure 20. Base class C

```
public aspect A1{
    pointcut pc(): call(void C.m());
    before(): pc()    {
        System.out.println("before A1...");    }
    before(): pc()    {
        System.out.println("before A2...");    }
    after(): pc() {
        System.out.println("after A1..."); }
    after(): pc() {
        System.out.println("after A2..."); }
}
```

Figure 21. The *A1* aspect

The output when *A1* (from Figure 21) is woven into C is:

```
before A1...
before A2...
This is the basic method.
after A1...
after A2...
```

This output is straightforward and easy to understand. However if we put them into two aspects as the Figure 22 shows, the output would be different.

```
public aspect A1{
    pointcut pc(): call(void
C.m());
    before(): pc()    {
        System.out.println("before
A1...")    }
    after(): pc() {
        System.out.println("after
A1...");}
}
```

```
public aspect A2{
    declare precedence: A1, A2;
    pointcut pc(): call(void
C.m());
    before(): pc()    {
        System.out.println("before
A2...");   }
    after(): pc() {
        System.out.println("after
A2...");   }
}
```

Figure 22. *A1* and *A2* aspects

The output is:

```
before A1...
before A2...
This is the basic method.
after A2...
after A1...
```

Notice that the order of the after advice in the output. This is different from the above output. Another problem with AspectJ is the precedence issue between the aspects that we focus on in this thesis. If we don't explicitly declare the precedence between the aspects, default precedence would be adopted, but sometimes this is not the programmer expects. So how to guarantee a

correct precedence is what aspectJ should improve in future, because for a very large project, such precedence is critical. We have provided a solution to this in this thesis.

2.3 Feature-Oriented Programming (FOP)

Feature-oriented programming is actually an extension of the object-oriented programming paradigm. Whereas object-oriented programming supports incremental development by subclassing, feature-oriented programming enables compositional programming overwriting with inheritance [30].

Feature Oriented Programming (**FOP**) is the study of feature modularity. While the term *feature* has a different definition in different papers, the main idea is that features are mapped one-to-one to modular implementation units (feature modules). Each feature contains a set of classes which are necessary to this feature and probably also to the other features. Feature implementations are actually very close to OO framework designs. This is demonstrated at the following part of the basic idea. A major contribution of object-oriented programming is the reuse by inheritance. Its success and its extensive use have led to several approaches to increase flexibility. **FOP** is a model for object-oriented programming which nicely generalizes inheritance and includes the extensions and new concepts [30].

FOP was developed to implement software incrementally in a step-wise manner. AHEAD (Algebraic Hierarchical Equations for Application Design) is an architectural model for **FOP** and a basis for large-scale compositional programming [2].

2.3.1 Basic Concepts

Consider a class as shown in Figure 23. The base class C has four member variables v1-v4. Apparently we can define C many ways by using inheritance with several classes. Figure 24 shows another possible way.

```
class C {
   int v1;
   int v2;
   int v3;
   int v4;
```

```
}
```

Figure 23. The base class

```
class C1 {
int v1;
}
```

```
class C2 extends C1{
  int v2;
}
```

```
class C3 extends C2{
 int v3;
}
```

```
class C extends C3{
   int v4;
}
```

Figure 24. Another definition of C

As you can see, a class extension can add new members. Actually it can also extend existing methods of a class. So C can be synthesized as C ● C3 ● C2 ● C1. From this idea, we can use a new keyword **refines** to represent Figure 24, as Figure 25 shows. **refines** means extension.

```
class C {
int v1;
}
```

```
refines class C {
  int v2;
}
```

```
refines class C {
 int v3;
}
```

```
refines class C {
   int v4;
}
```

Figure 25. Another representation type

So when we create a single class, we can use several steps to finish it. Now think if we want to finish a project, which is composed with a lot of classes, we can use same way for every single class. When considering the steps for every class, it is reasonable to use layers to group the steps between those classes, as Figure 26 shows. Thus **FOP** provides a good way to construct well-defined, encapsulated modules.

Layer 0 (the first step for all classes)
Layer 1 (the second step for all classes)
Layer 2 (the third step for all classes)
Layer n (the nth step for all classes)

Figure 26. A project with layers

FOP has advantages such as the improvement of the flexibility and scalability of the system. It also helps the designers separate requirements into different features; perhaps, too, it is well-suited to how designers work on a software system. Suppose a product line is finished, but

now we have to add or change some of the features of this product. People will hesitate to make changes to the existing code, for fear of breaking existing features. Using layers, one can cleanly encapsulate changes with minimal impact on existing code.

- **Design Rules**

The layers can be put in any sequence as the programmers want. For example, a project is composed as L4 ● L3 ● L2 ● L1, and it is understandable that a different composition order may be wrong, like L4 ● L3 ● L1 ● L2. Thus we need design rules to make sure a correct composition order is adopted.

Actually this is a fundamental problem in **FOP**. The use of a feature in a program can enable or disable other features. As introduced in the AHEAD tool suit, design rules are domain-specific constraints that define composition correctness predicates for features. Design rule checking (DRC) is the process by which design rules are composed and their predicates validated. AHEAD has two different tools for defining and evaluating design rules: `drc` and `guidsl`, though `drc` is a first-generation tool and `guidsl` is a next generation tool. However, both of them come from the same theory – the use of grammars to define legal sequences (i.e. compositions) of features. By this way, only legal composition sequence can be used for a given project.

However, we believe rule checking is insufficient to guarantee that the correct semantics is generated, because the compositions are more complex than they appear to be. We discussed this in the motivation part, and we provide a good solution to solve this.

Currently **FOP** is still an academic concept that has not yet been widely adopted by industry [32]. **FOP** is appropriate to implement such layered, step-wise refined architectures. During the evolution, software has to be adapted to fit unanticipated requirements and circumstances. This results in modifications and extensions that crosscut many existing implementation units in numerous ways, and this makes a big challenge to **FOP**.

Now there are few programming languages support **FOP**, and fewer mature frameworks that support **FOP**. This is a big obstacle to the future of **FOP**.

2.4 Instance-Oriented Programming (IOP)

IOP is an extension to **FOP**. **IOP** intends to combine **FOP** and the Model/View/Controller paradigm, and it provides a very straightforward GUI to view the layers and the corresponding software files. **IOP** also describes an elegant way to specify the concerns between features and provide an initial way to solve the precedence problem in **FOP**. The ACDK project is an ongoing project in WPI, to implement the idea of **IOP** [3].

In order to explain the idea of **IOP**, let us start with an example. This example is about the Klondike solitaire game. First Figure 27 shows the design of a game with six buildable piles and Figure 28 shows the executable program with this design. You can see there are six buildable piles in the game. Now, if you want to add another buildable pile, it is very easy to do so without touching the existing code but just add a layer of aBuildPile into the system. Figure 29 shows the design with seven piles, and Figure 30 shows the executable program with the seven piles.

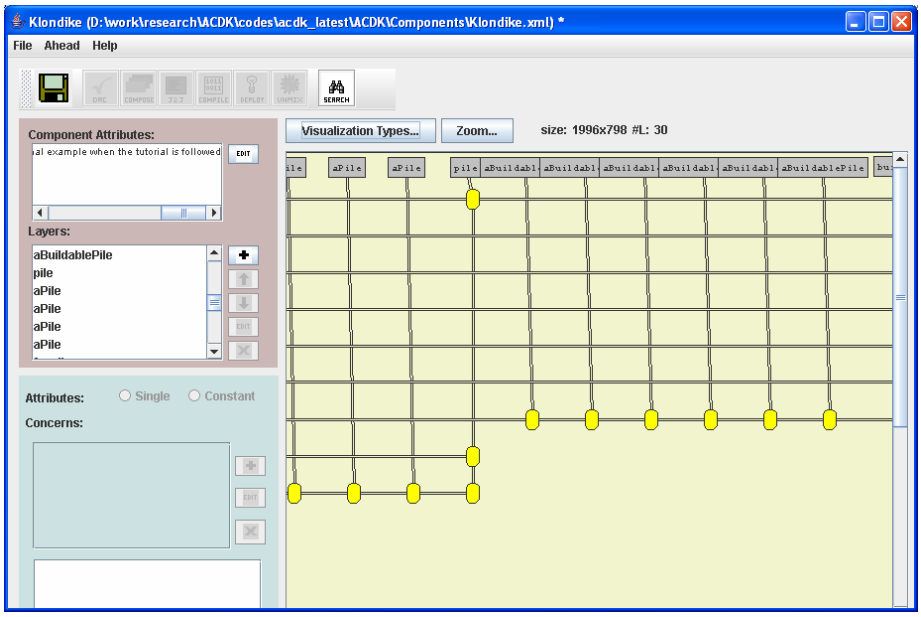

Figure 27. Design with six BuildablePiles

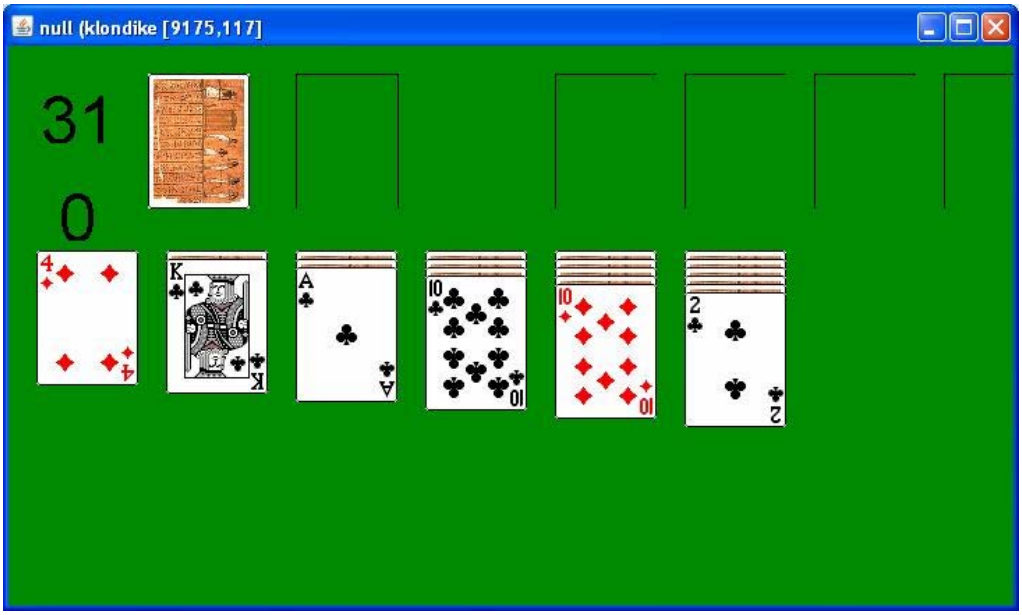

Figure 28. Executable program with six BuildablePiles

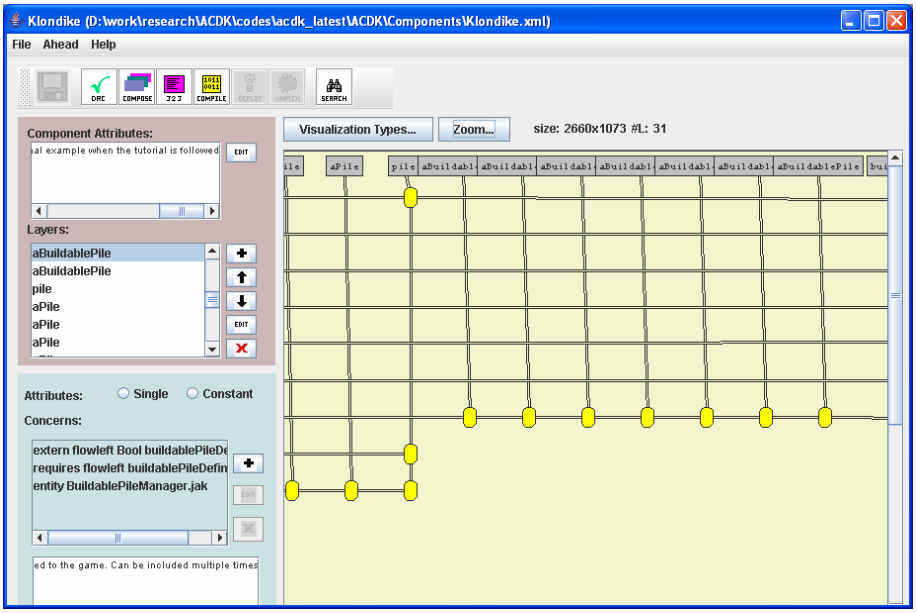

Figure 29. Design with seven BuildablePiles

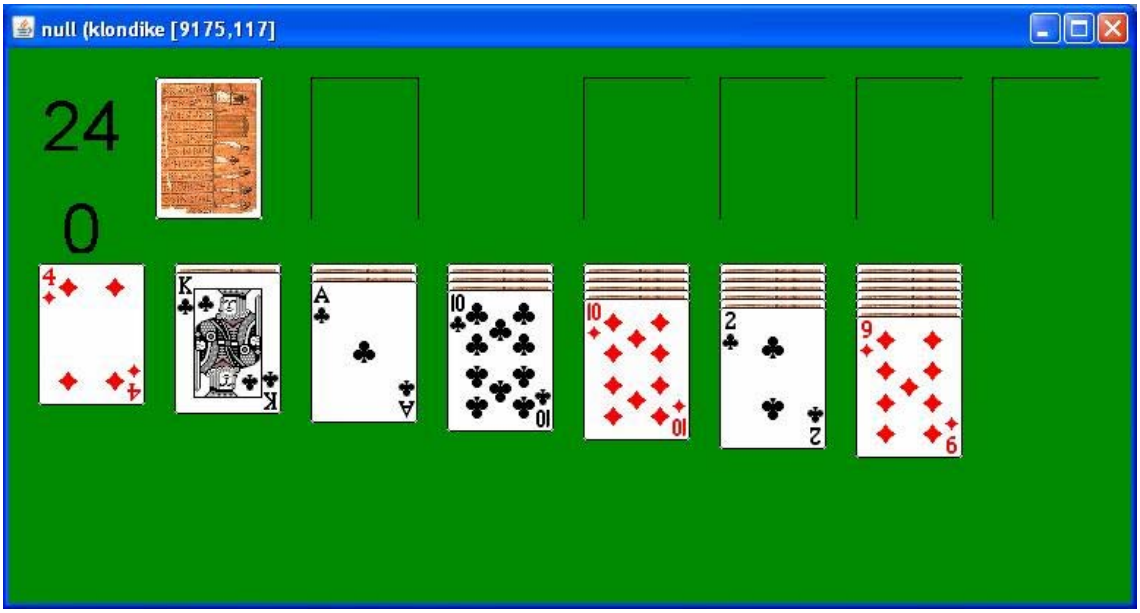

Figure 30. Executable program with seven BuildablePiles

From this example, it is clear that **IOP** focuses on the instances. We have to say that **IOP** is only adaptable to some systems, not to all the systems. For those systems which require multiple instances of the same thing, **IOP** is very useful and helpful. It makes a better management of the components. In the rest part of this thesis, we will not distinguish **FOP** and **IOP**, because they are same thing related to the dependencies that we focus on.

3 Analysis of Modular Dependencies

3.1 Class Dependencies in OOD

One of the characteristics of object-oriented software is the complex dependency that may exist between classes due to inheritance, association and aggregation relationships [34]. We may find out the dependencies between the classes by analyzing the data flow or control flow between them. We can also divide the dependencies into two groups. Given a class C, the two groups are defined as:

- $G_1(C)$ is the group of classes on which C depends statically, i.e. at compile time;
- $G_2(C)$ is the group of classes on which C depends dynamically, i.e. at run time;

In this thesis, we only focus on the dependencies caused by inheritance during compile time. The reason we focus on inheritance is that, when we consider the extension of an existing system, and inheritance is traditionally used to change the existing features of the system. The reason we only consider compile time is that we hope to detect the dependency conflicts during compile time, so it is not meaningful to consider the dependencies during run time, which may be caused by polymorphism. Thus we don't need to analyze the data flow between the classes because the relationship due to inheritance is very easy to identify.

3.1.1 Inheritance Dependencies

A subclass can add new attributes or methods, as well as change the methods in its superclass. What we care about is that how a method changes the method in the superclass.

There is an implicit ordering of functionality regarding methods in a subclass and overridden methods with the same signature in its ancestor classes. In some cases, a class **C** requires its subclass **SC** to override a particular method *m* (note that the current Java compiler allows the **@Override** annotation to be associated with a method, but this is only used to validate that the specific method in **SC** is actually capable of overriding a method in the base class **C**. It is not an error to omit this annotation when there is an override present, but it is an error to claim an override when no such relationship exists).

For the rest of our discussion, we assume that *m* is not abstract in either class. The relationship between **SC**.m and **C**.m has no fixed semantics, but is rather left to the discretion of the designer. There are several cases to consider:

- ***SC**.m replaces **C**.m* – The implementation of **SC**.m is used at runtime for all objects of class **SC**, and the original code found in **C**.m is never used.
- ***SC**.m incorporates **C**.m as a one-time invocation* – The implementation of **SC**.m provides some additional logic but makes a single call to **C**.m (via Java's **super**.m() invocation) to the original **C**.m code found in class **C**.
- ***SC**.m makes multiple calls to **C**.m* –This practice is generally discouraged because of the potential side effects that may result, yet there is no inherent ability for Java or C++ compilers to detect these situations.

When ***SC**.m incorporates **C**.m as a one-time invocation,* there are yet three possibilities to consider:

- ***super**.m() is the first statement in **SC**.m* – Termed the **pre** case as Figure 31 shows, **SC**.m first invokes the behavior as identified by the superclass **C** before it performs its actions.
- ***super**.m() is the last statement in **SC**.m* –Termed the **post** case as Figure 31 shows, **SC**.m first invokes its specialized logic and then requests the appropriate behavior to be invoked in the superclass **C**.
- ***super**.m() is the middle statement in **SC**.m* –Termed the **middle** case as Figure 31 shows, **SC**.m first invokes its specialized logic around the method in the superclass **C**.

pre	**middle**	**post**

`SC.m (params) {`
 	
`super.m(params);`
 	
`// now do extra`
 	
`}`
	`SC.m (params) {`
	`// do something first`
	
	`super.m (params);`
	
	`// do something next`
	`}`
		`SC.m (params) {`
		`// do something`
		
		`// now invoke parent`
		`super.m(params);`
		`}`

Figure 31, Three cases to consider

Note that even if method *m* returned a value, the above logic would still be the same.

3.1.2 Examples of Using Super From the JDK

Given the latest JDK 1.6 release (Fall 2006) has numerous examples where the method from a superclass is invoked from within the subclass.

```
/**
 * Processes events on this button. If an event is
 * an instance of <code>ActionEvent</code>, this method invokes
 * the <code>processActionEvent</code> method. Otherwise,
 * it invokes <code>processEvent</code> on the superclass.
 * <p>Note that if the event parameter is <code>null</code>
 * the behavior is unspecified and may result in an exception.
 *
 * @param       e the event
 * @see         java.awt.event.ActionEvent
 * @see         java.awt.Button#processActionEvent
 * @since       JDK1.1
 */
protected void processEvent(AWTEvent e) {
  if (e instanceof ActionEvent) {
     processActionEvent((ActionEvent)e);
     return;
  }
  super.processEvent(e);
}
```

Figure 32. ActionEvent example

Given the above code example as Figure 32 shows, drawn from **java.awt.Button**, ActionEvent events (present from the beginning JDK1.1) are handled differently from the other type of AWTEvents. This method clearly overrides the default implementation as defined in **java.awt.Component**, but doesn't invoke the **super**'s method for ActionEvent objects.

The **java.lang.Character** class provides an interesting example of invoking super as Figure 33 shows.

```
/**
 * Returns the standard hash code as defined by the
 * <code>{@link Object#hashCode}</code> method.  This method
 * is <code>final</code> in order to ensure that the
 * <code>equals</code> and <code>hashCode</code> methods will
 * be consistent in all subclasses.
 */
public final int hashCode() {
  return super.hashCode();
}
```

Figure 33. hashCode method in **java.lang.Character** class

This `hashCode` method seems irrelevant, but the **final** designation on the method clearly specifies that no future subclass can override this method to perform alternative behavior.

```
/**
 * Represents a list of values for attributes of an MBean.
 * The methods used for the insertion of javax.management.Attribute Attribute
 * objects in the AttributeList overrides the corresponding methods in the
 * superclass ArrayList. This is needed in order to insure that the objects
 * contained in the AttributeList are only Attribute objects. This avoids
 * getting an exception when retrieving elements from the AttributeList.
 *
 * @since 1.5
 */
public class AttributeList extends ArrayList  {

    ...

    /**
     * Sets the element at the position specified to be the attribute
specified.
     * The previous element at that position is discarded. If the index is
     * out of range (index < 0 || index > size() a RuntimeOperationsException
should
     * be raised, wrapping the java.lang.IndexOutOfBoundsException thrown.
     *
     * @param object  The value to which the attribute element should be set.
     * @param index  The position specified.
     */
    public void set(int index, Attribute object)  {
    try {
        super.set(index, object);
    }
    catch (IndexOutOfBoundsException e) {
        throw (new RuntimeOperationsException(e, "The specified index is out
of range"));
    }
    }
}
```

Figure 34. set method in **AttributeList** class

From the **javax.management.AttributeList** class as Figure 34 shows, the designers extend the base class **java.util.ArrayList** to store a set of **Attribute** objects; however numerous methods need to be protected to ensure proper behavior; below we show the *set* method implementation.

Designers are truly faced with numerous choices when designing the relationship between classes and their subclasses. Annotations can be used to capture the complex relationships between the various classes in a framework; if this information were found only in textual documentation, it could not play a central role in verifying that the proper design was followed. As an example, consider the intent of a designer to force the subclasses of a class **C** to provide an implementation for a specific method; in some cases, **C** can be defined as abstract. In other cases, however, there is no recourse. Here is what the **java.applet.Applet** class does as Figure 35 shows.

```
/**
 * Returns information about this applet. An applet should override
 * this method to return a String > containing information
 * about the author, version, and copyright of the applet.
 *
 * The implementation of this method provided by the
 * Applet class returns null.
 *
 * @return  a string containing information about the author, version, and
 *          copyright of the applet.
 */
public String getAppletInfo() {
 return null;
}
```

Figure 35. A method in **java.applet.Applet** class

If this method is not overridden by the **Applet** subclass, a potential problem exists if other framework classes expect this method to return a meaningful value. Clearly the design should be able to annotate this method so comprehensive analysis could detect potential errors in subclasses that choose not to override this method.

Another similar example is related to three classes: **java.awt.Rectangle2D**, **java.awt.Rectangle** and **javax.swing.text.DefaultCaret**. **Rectangle** extends from **Rectangle2D** and **DefaultCaret** extends from **Rectangle**. The two subclasses override the `equals` method but they use different ways to override. **Rectangle** calls the **super** method as its last statement; however **DefaultCaret** never calls the **super** method. Details on how they are implemented can be found in the JDK

source files.

We focus our attention on the Java programming language, although the ideas would be present in object-oriented languages such as C++ or even Perl. The Java language provides the **final** keyword to enable designers to restrict subclasses from overriding methods of a class (indeed, from preventing designers from even extending a class designated as final).

3.2 Aspect Dependencies in AOP

Aspectual dependencies and interactions, similar to the aspects themselves, are not confined to one development stage, but span the whole development cycle: from requirements to implementation. In view of the effects of the aspectual dependencies and interactions, considerable research and development work on these issues has been undertaken even since the inception of **AOP** and AOSD. A significant number of papers in journals and conferences have been published and several workshops including this theme have been successfully organized, showing that the field has a wide base of continuous research being done by established groups around the world.

Dependency covers the situation where one aspect explicitly needs another aspect and hence depends on it. Without the other aspect, the former aspect cannot perform correctly. A dependency does not result in a problem or erroneous situation as long as the aspect on which another one depends is ensured to be present and not changed [35]. To illustrate this situation, two simple dependencies are the following:

- In the context of security, authorization depends on authentication.
- In the example of TeleCom we mentioned earlier, *Billing* depends on *Timing*.

3.2.1 Analysis

Again, same as in **OOD**, We may find out the dependencies between the aspects by analyzing the data flow or control flow between them. There are already some papers discussed this, like [18]. However, in this thesis, we don't do that way. Instead we only consider the dependencies as long as the aspects share same joinpoints. In this situation, we treat them as being dependent on each other. The reason is that, we only focus on the dependencies which may cause the precedence issue. We know that advice is that basic unit to contain the changes. And advice uses joinpoints to

inject the changes, so joinpoints are the "evil source" for the precedence issue. If they don't share same joinpoints, the aspects will not be woven together so one can't generate the precedence issue. As long as we catch those dependencies with same joinpoints, we can resolve the precedence issue.

As we have already known, for a piece of advice, it has the following types.

- ✓ before
- ✓ around
- ✓ after
  - ■ after
  - ■ after returning
  - ■ after throwing

We may find that they are corresponding to the relationships between the subclass method and the superclass method as discussed in **OOD** part.

- ✓ For a piece of before advice, the **super.m()** is the last statement in the method of the subclass.
- ✓ For a piece of after advice, the **super.m()** is the first statement in the method of the subclass.
- ✓ And for a piece of around advice, the **super.m()** is the middle statement in the method of the subclass (as called proceed() in **AOP**).

The difference between **OOD** and **AOP** at this point is that, in **OOD** the order between subclasses and superclasses are fixed, that is, we know which must come first and which must come last; but in **AOP**, the order between the aspects are not fixed, that is, they are either defined by declare precedence statement, or left undefined. What is more, the subclasses at the same level in **OOD** will not be woven together to generate new code, the aspects do need to. So the final woven code in **AOP** may have several versions, and thus some wrong versions might be generated, and in this condition we call them conflicts.

Conflicts include the following two types:

- ■ One aspect totally depends on other aspects and a given order between that aspect and the other aspects must be followed. If such order is not followed or the depended aspects are removed, a conflict occurs..
- ■ One aspect that works correct in isolation does not work correctly anymore when it is

composed with other aspects. Thus this type of conflict occurs. An aspect influences the correct working of another aspect negatively, often because different (sub-)concerns will have conflicting requirements. Typically, the conflict can be solved by mediation between the two aspects, because – in a sense – they are complementary.

3.2.2 Examples of Conflicts

Conflicts are actually also "*dependencies*". They depend on each other to avoid conflicts. The TeleCom is a very good example for the first type of the conflicts – the billing aspect depends on .timing. Here we give examples for the second type.

We still use the TeleCom example. Suppose now the TeleCom system is working very well, but we will add some new features to that system. First we give an example which demonstrates that one aspect is added to the system and there is not any conflict. This example requires that logging should be added to log the starting time and ending time for every connection. An aspect called TimerLog is finished as Figure 36 shows.

```
public aspect TimerLog {

    after(Timer t): target(t) && call(* Timer.start())  {
      Logger.log("Timer started: " + t.startTime);
    }

    after(Timer t): target(t) && call(* Timer.stop()) {
      Logger.log("Timer stopped: " + t.stopTime);
    }
}
```

Figure 36. The *TimerLog* aspect

After this aspect is woven with the other aspects, the system is still working very well. Now let us have a look at an example that the new aspect is harmful to the other aspects, that is, a new aspect is added then the other aspects are not working. Suppose we need an aspect to reset the starting time and the ending time to zero after a connection is stopped. The aspect as Figure 37 shows is added to the system.

```
public aspect Reset {

    after(Timer t): target(t) && call(* Timer.stop()) {
        t.startTime = 0;
        t.stopTime = 0;
    }
```

```
}
```

Figure 37. The *Reset* aspect

This aspect is added and woven after timing but before billing. Thus the billing is always zero, and this is definitely not expected.

When conflicts occur, sometimes the programmers can find them immediately by some output, but they are not always so lucky. Some conflicts may only happen during some special conditions. So some tools are needed to guarantee that no conflicts occur in the system.

3.3 Feature Dependencies in FOP

From the background part, we know that **FOP** comes from the idea of inheritance in **OOD**. So the dependencies in **FOP** have something in common with the dependencies in **OOD**, regarding the relationship between the classes. Also, we know that **AOP** and **FOP** both generate final code by weaving the aspects or features together, so the dependencies in **FOP** have something in common with the dependencies in **AOP** too.

One layer is actually one feature in **FOP**. One layer includes some classes. So the feature dependencies are actually the dependencies between groups of classes. At the same time, we know that there is always an order between the layers. So there are two elements which dominate the dependencies between the classes and dominate the final semantics:

1. The layer order between the layers;
2. The position of the `Super` call ocurrs.

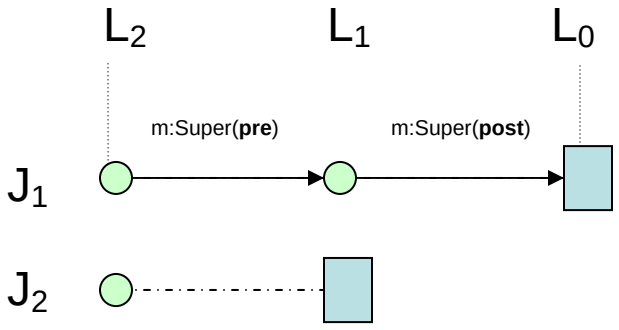

Figure 38. An example equation

On one level, a shallow form of precedence appears to exist within **FOP** because of the ability to specify a system as an equation of the composition of a series of layers. However, the underlying situation is more complex, as we showed in the example. Consider the following system S = L2 ● L1 ● L0 as Figure 38 shows. In this equation, the "base" layer L0 contains the definition of a Java class file (J1) with a single method m. Layers L1 and L2 refine the m method of J1. As the above example shows, within **FOP**, a layer can refine an existing layer. For example, using the AHEAD toolset, L1 could refine J1 as Figure 39 shows.

```
refines class J1 {
    public void m() {
      ... // execute refinement code
      Super().m();
    }
}
```

Figure 39. Possible refinement with AHEAD

The terminology "m:Super(post)" means that within the L1 refinement, method m invokes its "Super" method after it has accomplished its work. Another alternative, would be "m:Super(pre)" where the refinement first invokes the "Super" method before it does any work. The third alternative, is "m:Super(middle)", which means the refined method invoke "Super" method in the middle of the code. Another alternative is "m:Super(null)" which means that the refined method completely overrides the implementation of m within its class.

In most cases, the refined method should only invoke its "m:Super()" method exactly once, otherwise the semantics of the original method call may change in unexpected ways. Prehofer describes this exact property in his paper on Conservative Weavings [10]. We extend this same restriction a little bit that we allow a total replacement, which means the refined method does not call Super at all. Thus we classify the method refinement as having two parts: before-super and after-super. The "before-super" part contains the statements executed before Super is invoked, while "after-super" contains the statements executed after Super has completed. Apparently "before-super" and "after-super" could be empty when the call to Super is m:Super(pre) or m:Super(post). Again, we may find that this is same thing as the advice we talked about in **AOP** part above. m:Super(pre) is corresponding to the after advice in **AOP**. Due the clear corresponding relationship, we won't mention the others at this part.

Indeed, given the picture of the equation file earlier, it is only with the annotations on the horizontal edges that one knows the overall semantics of how the method invocation m has been changed. For example, the equation for J1 in the Figure 40 will produce the following execution sequence:

L2.before-super ; L1.before-super ; L0 ; L1.after-super ; L2.after-super

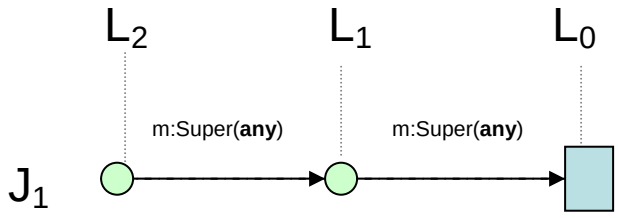

Figure 40. An example equation

4 Solutions to the Precedence Issue

Given the modular dependencies we have investigated within these design methodologies, we provide a way for each methodology to better manage the dependencies. We also claim that it is possible to come up with a uniform model (see Section 4.4) that captures the common features that are present in all methodologies. For this research effort we enable designers to annotate the individual modules with annotations that are then analyzed to determine if the resulting composition is consistent.

Definition: A *consistent* composition satisfies all known module annotations.

Because the space of annotations is universal, we restrict our attention to the ability for designers to specify ordering constraints among the various design modules.

4.1 OOD

Though there is not any common precedence issues in the **OOD** paradigm, from the dependencies we analyzed above, we can see that the relationship due to inheritance is complex and there is not any tool to help clarify the relationship.

To support the complex needs of Object-oriented designers, we need a set of annotations to clarify the intent of the designer when complex class relationships are formed. We have designed a set of annotations that are appropriate when both **C** and **SC** have concrete implementations of method *m*. An annotation always exists within a specific class, known as the *target* of the annotation. In the situations below, sometimes the target class is the subclass **SC** in the relationship; at other times, it is the superclass **C**. In all cases, the context is clear.

The first four annotations are associated with the target subclass method **SC**.*m*:

- **@Before** – Force the subclass method **SC.***m* to invoke *super.m()* before any of its in its execution; not doing so would be an error
- **@After** – Force the subclass method **SC.***m* to invoke *super.m()* as the last statement in its execution; not doing so would be an error
- **@Override** – This annotation is already present in the JDK standard. We leave it as it is, and choose to interpret it simply as a clarification that the given method *m* is truly intended to override the functionality of a method in its super class. However, this knowledge is not enough to capture the complex relationship (even with **@Before and @After**)
- **@Replace** – This annotation prevents the *super.m*() method from being invoked within the subclass method. Note that we can't default to this behavior when the annotation is not present because of the ambiguity of the @override annotation

There are annotations that would be useful in the superclass **C** as well. These are specification annotations to be applied to all subclasses (recursively) of the class. For each of the four annotations above, there is a corresponding **@DenyAction** and **@MustAction** that can be present in the target superclass:

- **@MustOverride** – Declare that a subclass must override a particular method in **C**. In Java (and other object-oriented languages) a method can be declared abstract (or *virtual* in C++) which essentially forces concrete subclasses to override the method
- **@DenyOverride** – Make it impossible for a subclass to override a particular method. In Java, this can be accomplished by using the **final** modifier for a method

- **@MustReplace** – Force the subclass method *m* to have **@Replace** as its annotation and prevent any attempt to simply inherit as is the default **C** *m* implementation. We require this capability because it is not simply sufficient to assume that **C** could design *m* as abstract (i.e., it could be required for objects of class **C**)
- **@DenyReplace** – Make it impossible for a subclass to override and replace a specific method. Subclasses must include an invocation to super.m in their implementations of *m*

- **@MustBefore** – Force subclasses to only have **@Before** annotations. This annotation declares implementations of subclass "**SC**.m" must execute before the invocation of **C.***m*
- **@DenyBefore** – Make it impossible for subclasses to have **@Before** subclass behavior where the method **C**.m is invoked via super before the execution of **SC**.m's special code
- **@MustAfter** – Force subclasses to only have **@After** annotations. This annotation is used to say "**C**.m" must execute after the invocation of any of the implementations of *m* in any subclass **SC**
- **@DenyAfter** – Make it impossible for subclasses to have **@After** subclass behavior where the method **C**.m is invoked via super after the execution of **SC**.m's special code

Put together, this set of annotations is complete and we can put together a matrix that shows the compatibility between annotations in the superclass **C** and the subclass **SC**. Note that these compatibilities are transitive to ancestor classes of **C**.

	Override		Replace		Before		After	
	@must	**@deny**	**@must**	**@deny**	**@must**	**@deny**	**@must**	**@deny**
@Before	YES	NO	NO	YES	YES	NO	NO	YES
@After	YES	NO	NO	YES	NO	YES	YES	NO
@Replace	YES	NO	YES	NO	NO	YES	NO	YES
@Override	YES	NO	YES	YES	YES	YES	YES	YES

Given these annotations, then, the above table shows the allowed compatible annotation pairs. **@MustReplace** is not compatible with **@Before**, for example, while **@DenyBefore** is compatible with **@After**.

We now define an algorithm to analyze the set of object-oriented design annotations to determine if the design is consistent with the annotated constraints. We omit from consideration the inheritance relationships inherent in the use of **java.lang.Object** as the implicit superclass for all classes in Java. The only clarification we would like to point out is that often there are relationships between methods of the same class that cannot easily be expressed by means of the

annotations we have proposed. The contract for **java.lang.Object** demands that "If two objects are equal according to the *equals(Object)* method, then calling the *hashCode* method on each of the two objects must produce the same integer result.". Capturing this complex dependency would require recourse to defining pre- and post-conditions for the methods in **SC** and **C** respectfully. Thus, if **SC** chooses to override *equals(Object)*, then there should be a corresponding override of the *hashCode* method. In our model, we have no place for relationships between annotations found in the same class. This topic is left for future work. For this algorithm to work, we assume the existence of a tool that evaluates source code to determine the following functions:

- *int invokesSuper (Class c, String m)* – the number of times method *m* in class *c* invokes *super.m()*
- *Type typeSuper (Class c, String m)* – Determines the way that method *m* in class *c* invokes *super.m()*; returns either **pre** (the invocation to *super.m* is the first statement), **post** (the invocation to *super.m* is the last statement), **middle** (code exists both before and after the invocation to *super.m*), or **undef** (the method *m* in class *c* contains control flow that makes it impossible to clearly classify the use of *super.m*).

An algorithm has been given for the consistence checking below.

ALGORITHM: **ConsistencyChecker**. Determine if the annotations defined in the set of classes is consistent with the implementation of the classes.

INPUT:

The set ***CL*** identifies the classes. We also define the inheritance relationship ***IN*** between these classes where ***IN*** is drawn from the set ***CL*** × ***CL***. If class **SC** extends **C** then (**SC**, **C**) ∈ ***IN***. Note that ***IN*** ignores the implicit extensions of **java.lang.Object** by all Java classes.

PRE-PROCESSING:

Construct the directed multigraph G = (V, E) where V is the set of vertices and E contains a set of directed edges between these vertices. Each node *n* in V represents a class **C** from ***CL***, thus |V|=|***CL***|. For each element (**C, SC**) ∈ ***IN***, create a set of labeled edges S = {*e* = (u, v) | u is the node representing **SC** while v is the node representing **C**}. For each edge *e* ∈ S, *e.method* refers to the method *m* defined in **C** which is overridden by a method *m* in **SC**, *e.source* represents the annotations associated with **C.***m*, and *e.target* represents the annotations associated with **SC**.*m*. The edges in S are added to E. Clearly the time to construct the graph is O(|***IN***|) and the graph is acyclic because Java supports only single inheritance.

STEPS:

Iterate over all classes in C and traverse links back to the superclass, building up the allowed annotations in the ultimate relationship, stopping and declaring as errors when a specific annotation is invalid given the known compatibilities.

Now we show how this algorithm works. First we get the classes and inheritance information as Figure 41 shows. Then we construct the directed multigraph based on that information, as Figure 42 shows. Each node *n* in **V** represents a class *C* from **CL**. For each (**SC, C**) ∈ **IN**, construct edge *e* in E representing a method in *C* that is overridden by a method in **SC**, where

e.source = annotations in **C**

e.target = annotations in **SC**

Clearly the time to construct the graph is O(|**IN**|) and the graph is acyclic since classes can not inherit from each other recursively. The graph is also simple since Java only allows single inheritance.

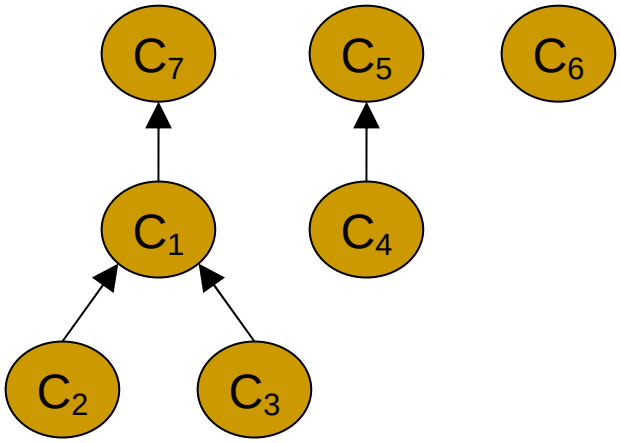

Figure 41. CL and **Inheritance** relationship

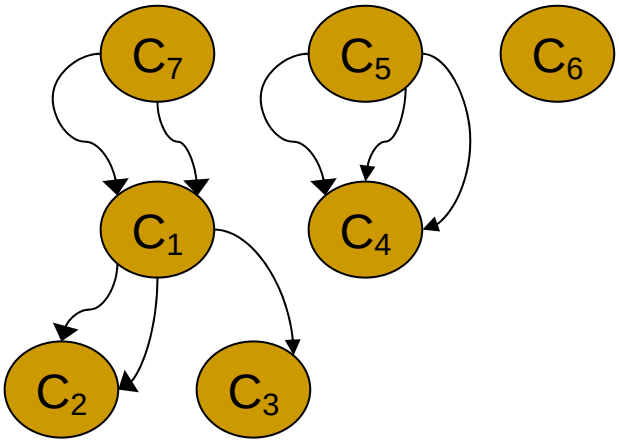

Figure 42. The directed multigraph

Then the last step is to iterate over each node in Figure 42 to find out the errors given the known compatibilities.

4.2 AOP

By analyzing how and why the precedence issue may cause the semantics problem, we decide to use annotation to solve the problem. Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program [33]. Annotations complement javadoc tags. In general, if the markup is intended to affect or produce documentation, it should probably be a javadoc tag; otherwise, it should be an annotation. You may have known that there are three annotation types that are predefined by the language specification itself: **@Deprecated**, **@Override**, and **@SuppressWarnings**. So now we will create our own annotations.

Basically for every piece of advice, the programmer can use annotations to specify where the advice should be invoked during the final weaving process. If a wrong order is used against the corresponding annotation, an error would be reported.

Here are the annotations we have defined so far. All the annotations' *RetentionPolicy* is *RUNTIME*, which means that those annotations will be kept by the compiler during runtime.

@BeforeFirst	The advice must be executed first before the refined method is executed.
@Before	The advice can be executed anywhere before the refined method is executed.
@BeforeLast	The advice must be executed as the last one before the refined method is executed.
@AroundFirst	The advice must be executed first before the refined method is executed, and must be executed the last one after the refined method is executed.
@Around	The advice can be executed anywhere around the refined method.
@AroundLast	The advice must be executed last before the refined method is executed, and must be executed the first one after the refined method

	is executed.
@AfterFirst	The advice must be executed first after the refined method is executed.
@After	The advice can be executed anywhere after the refined method is executed.
@AfterLast	The advice must be executed as the last one after the refined method is executed.
@Before(*Aspect***)**	The advice must be executed before related advice (same joinpoint) in another aspect.
@After(*Aspect***)**	The advice must be executed after related advice (same joinpoint) in another aspect.

Basically we can go through every method in the classes, find out all the advice that refine that method, by going through all the advice in all the aspects, then find out the possible execution orders of the advice, and during the finding process, we compare the advice execution position against its annotation. If we find out any conflict error, the find process will stop immediately and report this error.

We have two types of the result for the checking. One is Warning, which does not affect the semantics, but we think it is a potential problem to impact the semantics; the other one is Error, which makes the semantics wrong. So when we find out some warnings, we don't stop the finding process, but if we find out any errors, the process should be stopped immediately.

For warnings, there are two types:

Annotation is missed.	If there is not any annotation for a piece of advice, a warning is generated for that advice.
No precedence is declared between two dependent aspects.	If we can not find a precedence relationship between two dependent aspects, a warning is also generated.

For errors, there are five types:

Annotation conflicts with its own advice.	Represents the error when a wrong annotation is used on a piece of advice, for example, a **@BeforeFirst** annotation is declared on an after advice. Notice here that the relative annotations like **@Before**(*Aspect*) or **@After**(*Aspect*) can be put on any type of advice.
Not the first advice is executed.	One piece of advice is declared to be the first to be executed, but actually not. This includes **@BeforeFirst**, **@AfterFirst** and **@AroundFirst**. Notice for **@AroundFirst**, the pre-execution and the post-execution are different. For pre-execution, the advice must be executed first, but for post-execution, it must be executed last.
Not the last advice is executed.	One piece of advice is declared to be the last to be executed, but actually not. This includes **@BeforeLast**, **@AfterLast** and **@AroundLast**. Notice for **@AroundLast**, the pre-execution and the post-execution are different. For pre-execution, the advice must be executed last, but for post-execution, it must be executed first.
Not before a given aspect.	A piece of advice is declared to be executed before the other advice in another aspect, but actually it is not guaranteed to be before it. This is only for relative annotation.
Not after a given aspect.	A piece of advice is declared to be executed after the other advice in another aspect, but actually it is not guaranteed to be before it. This is only for relative annotation.

An algorithm is given below on how to check the consistency.

ALGORITHM: **ConsistencyChecker**. Determine if the annotations defined in the set of aspects is consistent with the implementation of the aspects and the base classes.

INPUT:

The set of the classes and aspects, and the declared precedence relationship between the aspects, together with the annotations and corresponding advice. The input may look like as Figure 43 shows.

STEPS:

1. Check the conflicts between the aspects. If two aspects are dependent on each other but there is not any precedence relationship declared between them, warnings should be reported.
2. Check whether the annotations are compatible after the aspects are woven together with the classes. This checking process starts from all the base methods in the base classes. For each base method, we go through all the aspects to find out all the advice that try to change the method, then based on the precedence between the advice, we build the final execution order for that method. During the building process, we check the annotations to make sure that they are always correct when a piece of new advice is added to the execution order. For example, if the first advice is declared to be executed **@BeforeFirst**, but a piece of new advice can be added to the head of the execution order (even that advice can be added somewhere else, but as long as there is a possibility), a conflict should be reported and the whole checking process quits.

Based the above, a design model is given. The annotation diagram is shows at Figure 44. The result errors and warnings diagram is shown at Figure 45. The main classes diagram is shown at Figure 46.

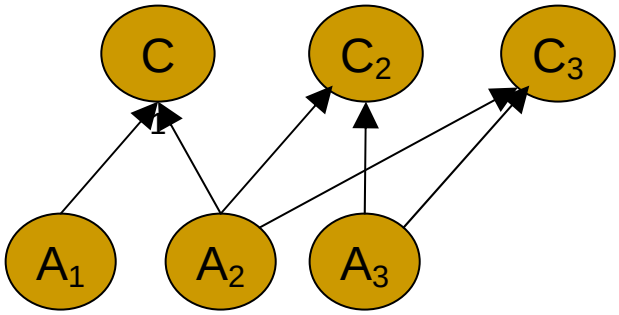

Figure 43. Set of classes and aspects

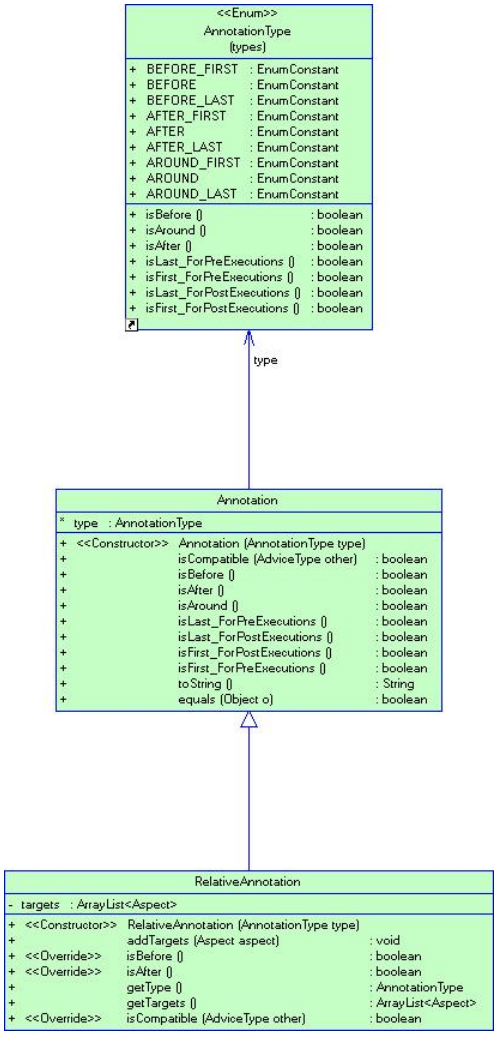

Figure 44. The annotation diagram

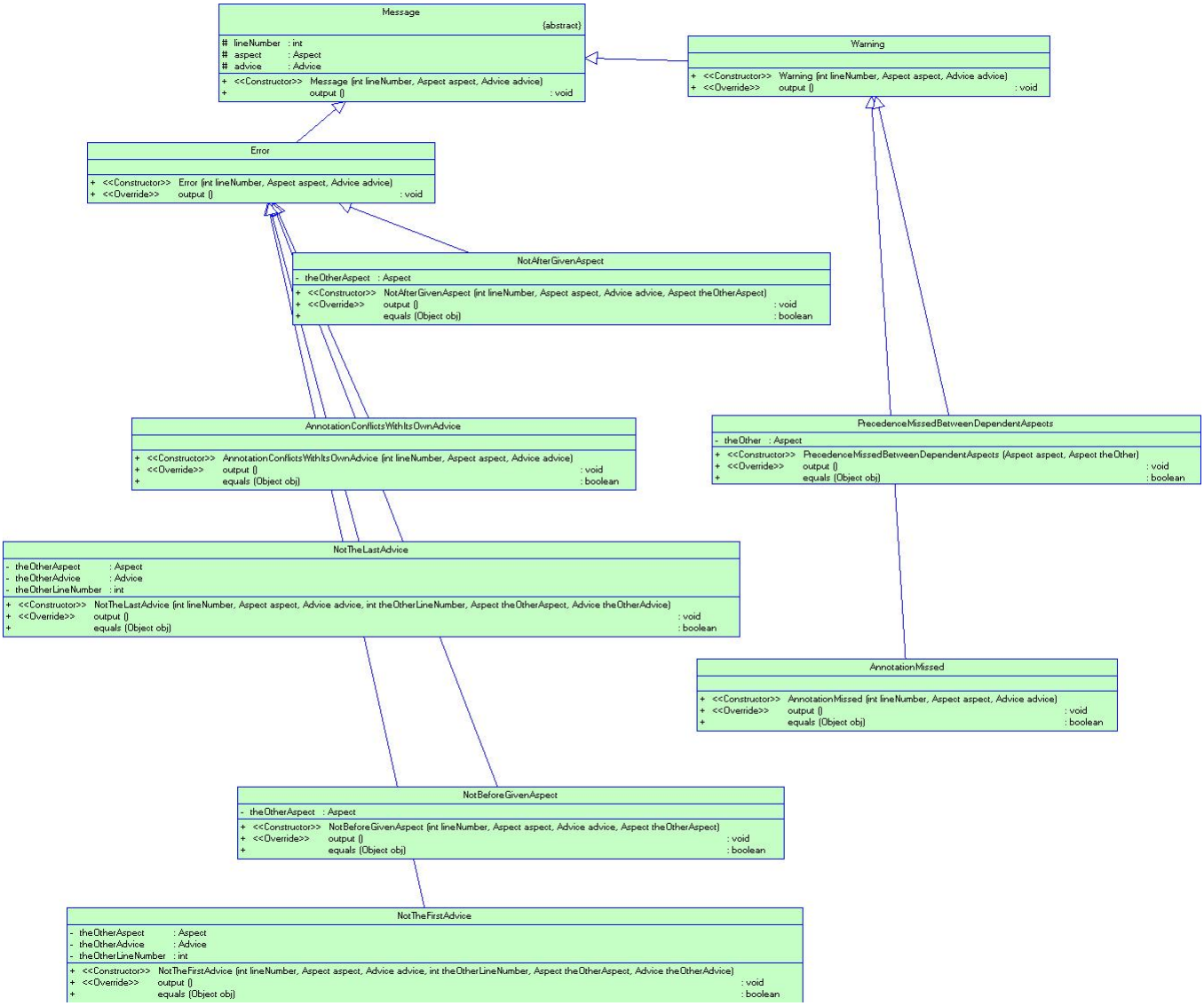

Figure 45 The reported errors/warnings diagram

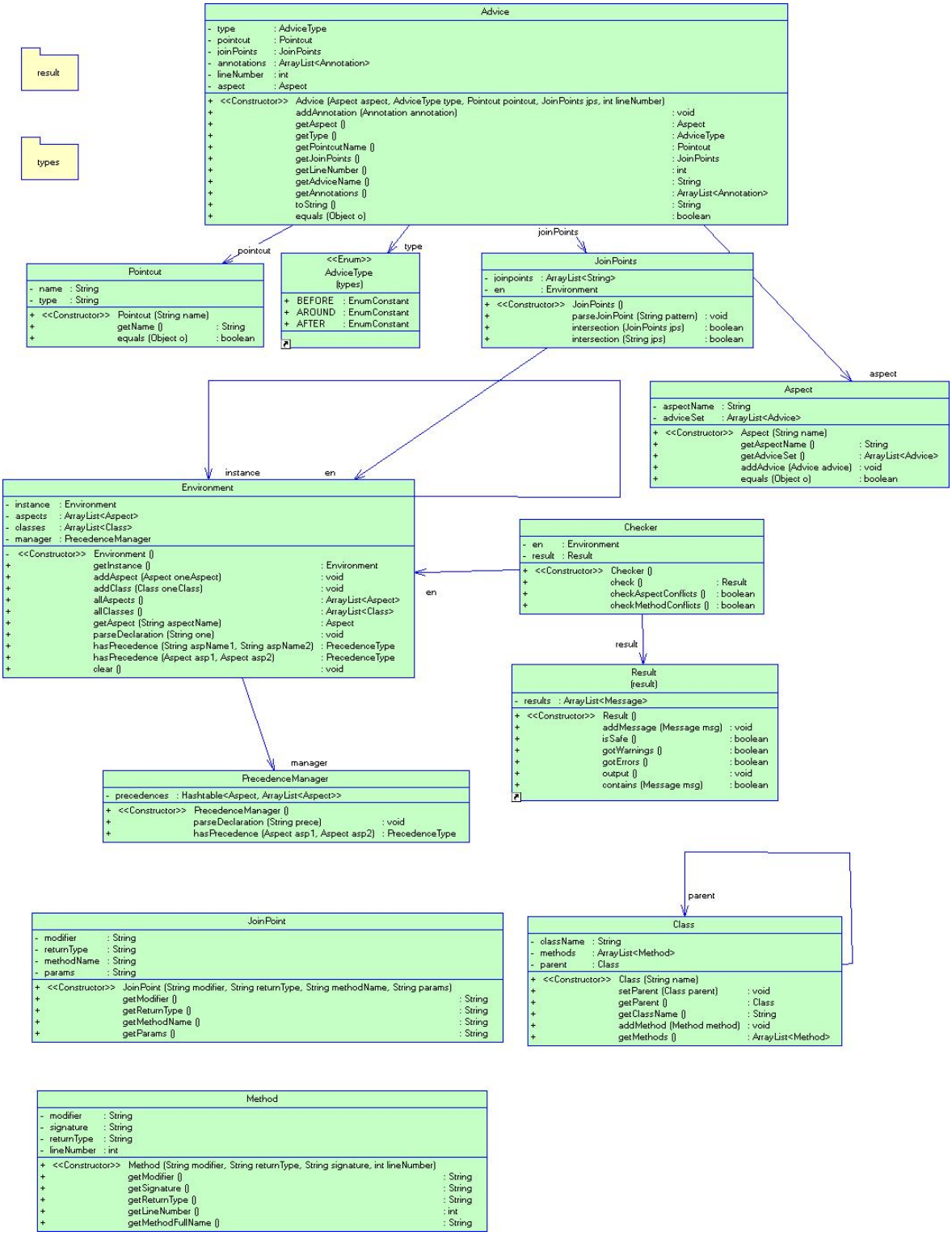

Figure 46. Main classes diagram

4.3 FOP / IOP

When considering the precedence problem **FOP** is actually very similar to **AOP**. However, concern is a new concept in **FOP**, which means some layer may require something to be provided on its left or right. A before advice in **AOP** can be implemented in **FOP** by just invocating the Super call at the last line of the refined method. Similarly an after advice can be implemented in **FOP** by putting the `Super` call at the first line of the refined method. For around advice, the `Super` call should be put inside the refined method. The only difference lies when building the execution order. In **AOP**, the final execution order is determined by the precedence of the aspects and the type of the advice. The precedence can be specified explicitly, but sometimes some of them are not specified, so some random precedence can be adopted between some aspects. In **FOP** the execution order is determined by the order of the layers and the position of the `Super` calls. The order of the layer is always specified.

We argue that in **FOP**, a `Super` call must be invocated exactly one or zero times. If a `Super` call is not invoked, there is a potential danger in breaking the relationship with the parent methods, but maybe this is sometimes wanted, though not common. If a `Super` call is called multiple times, a wrong and complex semantics might be used and this is not expected. Actually in this case a typo might be conducted. [10] claims same idea here. Note that the `Super` call over here is only about the call to its own parent method, and one time means that the `Super` call is called for one time during one execution, but maybe next time the `Super` call is at another location, but still only called for one time.

From this analysis, we decide to use same annotations as in **AOP** with a little change. And those annotations are also the basis for the unified model. We need to add a new annotation called **@Replace**. This annotation is used not common.

@Replace	The new method totally replaces the super method. The Super is not invocated.

We validate the **FOP** project in four steps:

1. Validate the concerns are satisfied.

2. Validate the annotations are compatible with their own code.
3. Validate the annotations are still correct after weaving the layers together.
4. Validate the relative annotations.

During validating the concerns, we check all the required concerns one by one, and when we find one of them is not satisfied, we quit the validating and report the error. If all of the concerns are satisfied, when validate that the annotations are compatible with their own code. For example, a **@BeforeFirst** annotation should have code which uses Super call at the last line, and **a @Around** annotation should have Super call inside the code. Then we move on to check whether the final execution order is correct based on the annotations. When building the execution order, we start from the base layer to find out all the normal methods (not refined method), then we go through all the other layers and find out the refined method on those normal methods, by considering the position of the Super calls, we can build the final execution order for the given normal method. During the building process, we check with the annotations on the methods to see whether they are satisfied. We quit the checking process as long as we find any error.

In the solution, we didn't mention Jak files. We think that is not necessary to be mentioned in the solution, because we know every layer contains a set of classes, including the normal classes and the refined classes. Jak files are just used to contain the classes, and they are not related to the execution order. So it is reasonable to skip the idea of Jak files.

Figure 48 shows the class diagram related to concerns. Figure 49 shows the similar class diagram as in **AOP** for the annotations. Figure 50 shows the results diagram. From the diagram we can see the following errors are used and reported. Figure 51 shows the main classes diagram.

Required flowleft is not satisfied.	One layer requires a flowleft but no layers on its right can provide that flowleft concern.
Required flowright is not satisfied.	One layer requires a flowright but no layers on its left can provide that flowright concern.
Annotation conflicts with the code of the method	Annotation is not correctly used on the method. This is caused by the position of the Super call.
Not correct Super call times.	The Super call should be invocated for exactly one time. Any wrong invocations will be

	reported with this error.
Not executed pre-first.	A method is annotated to be executed as **@BefireFirst** or **@AroundFirst**, but this is not guaranteed for the pre-part.
Not executed pre-last.	A method is annotated to be executed as **@BefireLast** or **@AroundLast**, but this is not guaranteed for the pre-part.
Not executed post-first.	A method is annotated to be executed as **@AfterFirst** or **@AroundLast**, but this is not guaranteed for the post-part.
Not executed post-last.	A method is annotated to be executed as **@AfterLast** or **@AroundFirst**, but this is not guaranteed for the post-part.
Not before given layer.	This is related to relative annotations. One method is annotated to be executed before the same method in another layer, but this is not guaranteed.
Not after given layer.	This is related to relative annotations. One method is annotated to be executed after the same method in another layer, but this is not guaranteed.
Not replacing the method.	A method is annotated to replace the super method but actually this is not the case.

The algorithm to validate the system is shown below.

ALGORITHM: **ConsistencyChecker**. Determine if the annotations defined in the set of refined classes are consistent with the implementation of the refined classes and the base classes.

INPUT:

The set of layers, and the precedence relationship between the layers. The layers include the corresponding base classes and refined classes, together with the concerns and annotations. Figure 47 shows a demo of the input, where *C* means regular class while *RC* means refined class on *C*.

STEPS:

1. Check the concerns between the layers are satisfied.
2. Check whether the annotations are compatible with their own actual code.
3. Check the non-relative annotations are compatible with the whole system. Same as in **AOP**, we build the execution order for every base method in the base classes. The process starts from the base layer. For every layer, first find out every base method, then for every base method, we go through the other layers on the left to find out all the refined methods on them. Thus an execution order can be built together with the precedence relationship between the layers and the position where Super is invocated. During the building process, annotations are checked with their refined methods' position in the execution order. If a conflict is detected, the whole checking process would quit and an error would be reported.
4. Check all the relative annotations. Same as checking the non-relative annotations, we check every layer from right to left and find out all the relative annotations then we compare this layer with the relative layer and the position of the Super is invocated, so as to find out that whether the relative annotations can be satisfied.

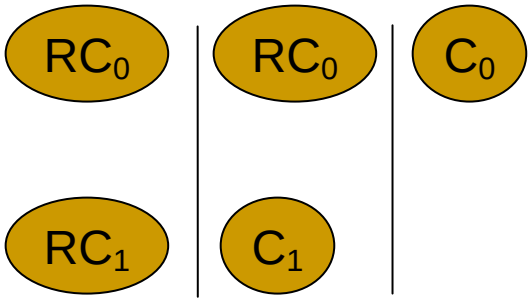

Figure 47. Example of input

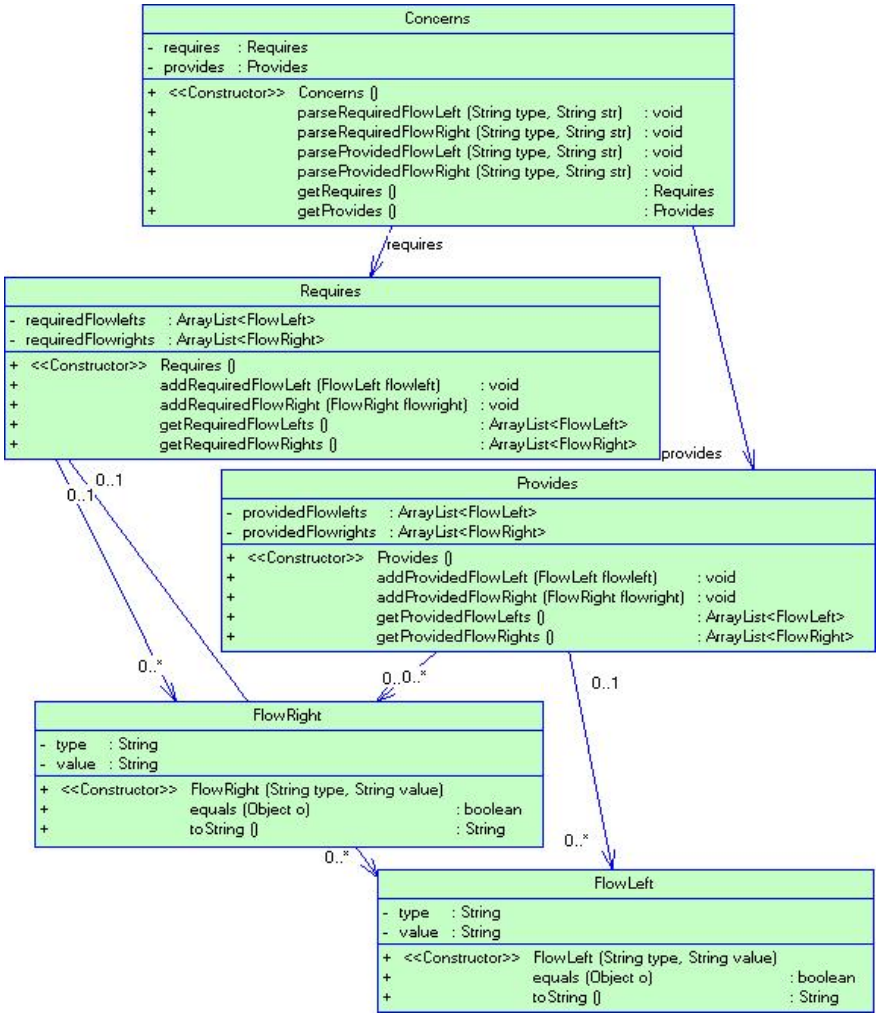

Figure 48. Concerns diagram

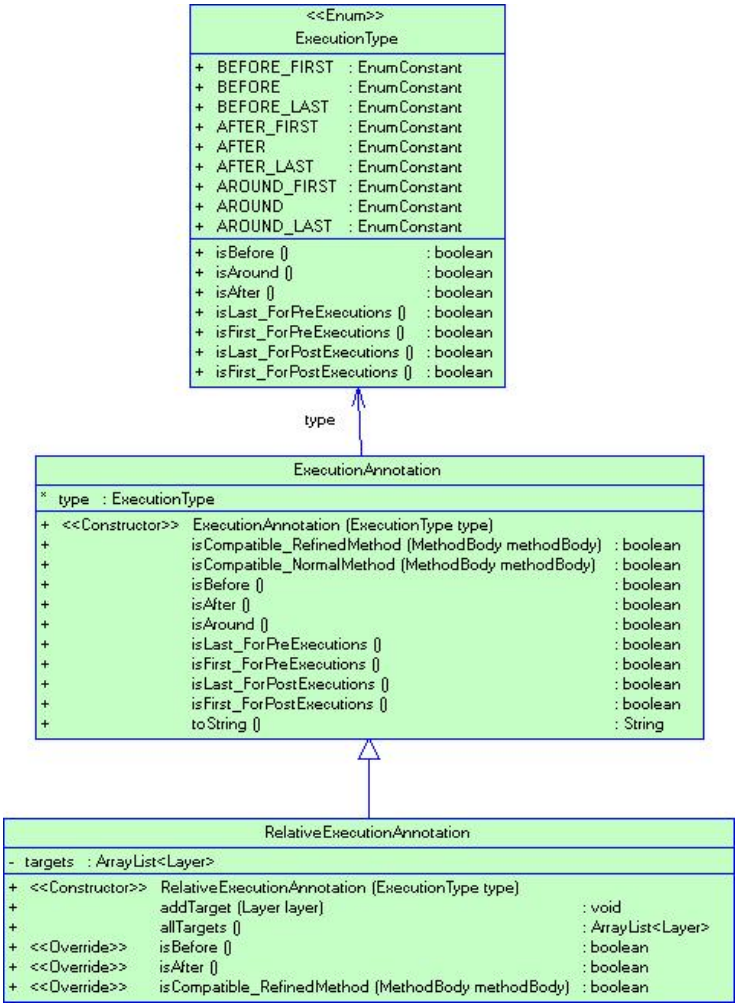

Figure 49. Annotations class diagram

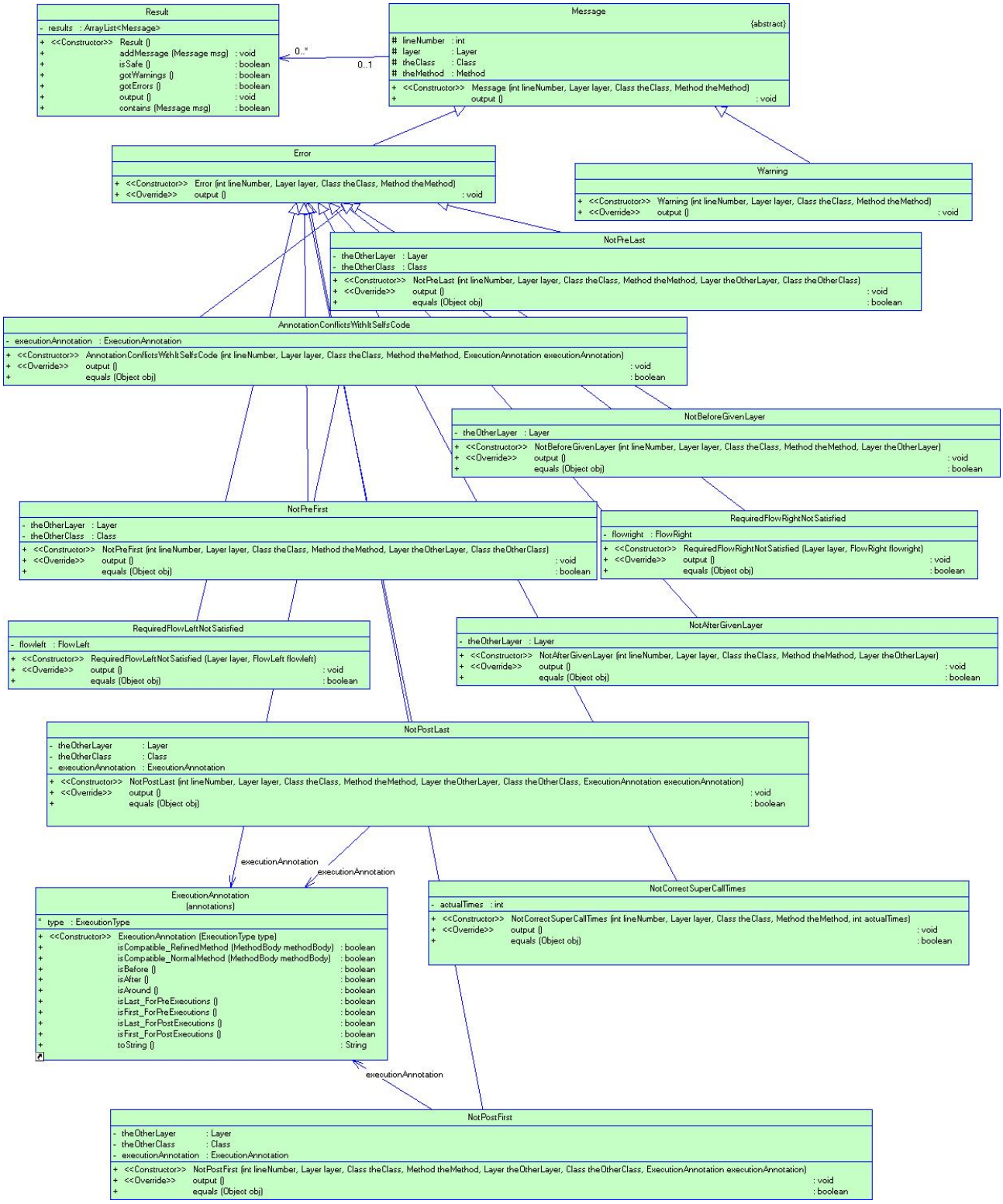

Figure 50. Result diagram

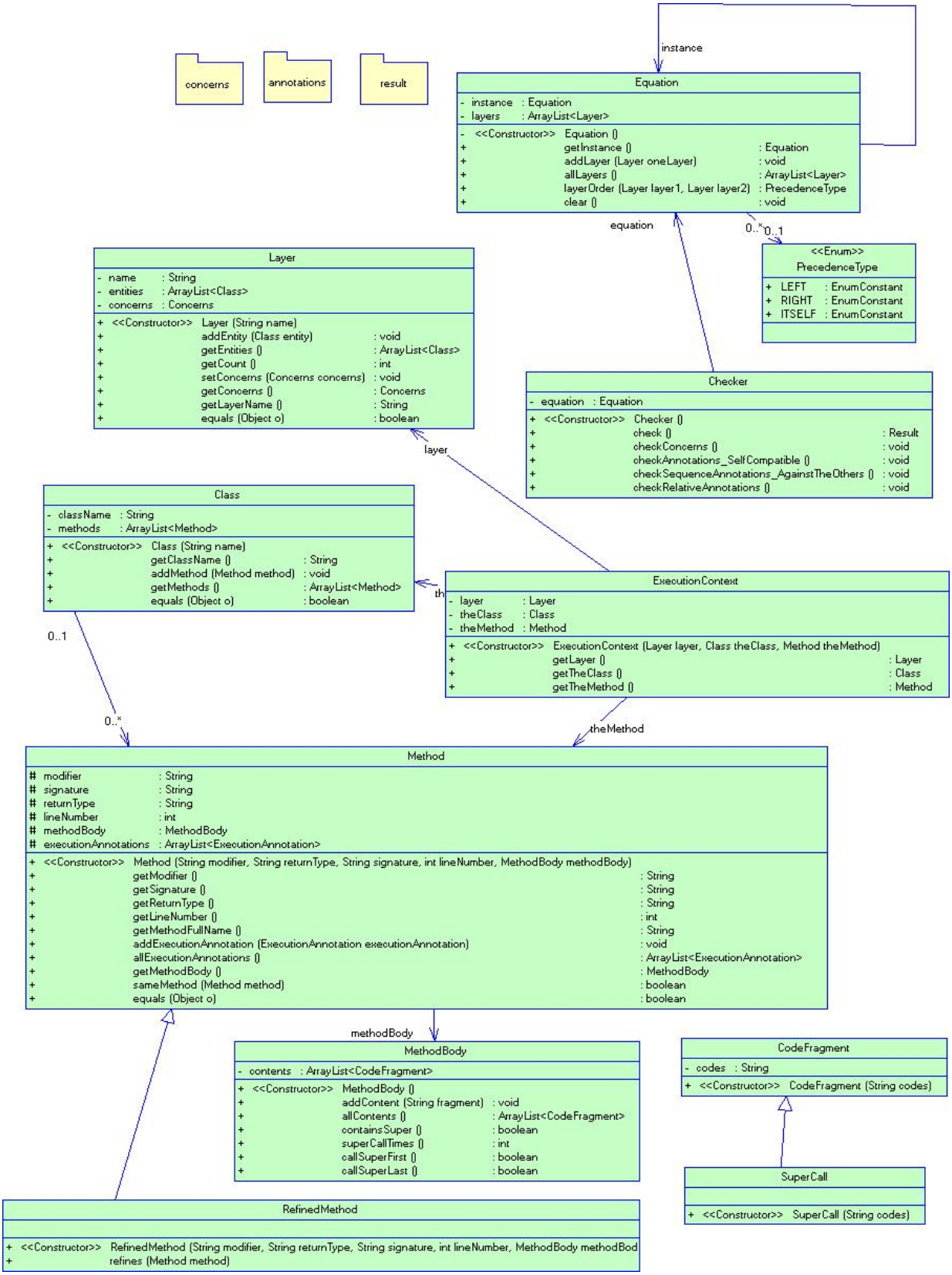

Figure 51. The main class diagram

4.4 The Unified Model

From all the discussions above, we can see that there is a core set of concerns in common between the paradigms, especially between **AOP** and **FOP** regarding the precedence issue.

We decide to adopt the following annotations as the unified model. In the unified model, we treat the layers or aspects as groups. And for **OOD**, the groups should be done based on the class hierarchy. For example, if two classes are extending the same parent class, they should be in the same group and they will be finally grouped like layers in **FOP**. We assume that this is already done by tools. So we can treat **OOD** as **FOP** in our model.

@BeforeFirst	The new code must be executed first before the basic method is executed.
@Before	The new code can be executed anywhere before the basic method is executed.
@BeforeLast	The new code must be executed as the last one before the basic method is executed.
@AroundFirst	The new code must be executed first before the basic method is executed, and must be executed the last one after the basic method is executed.
@Around	The new code can be executed anywhere around the basic method.
@AroundLast	The new code must be executed last before the basic method is executed, and must be executed the first one after the basic method is executed.
@AfterFirst	The new code must be executed first after the basic method is executed.
@After	The new code can be executed anywhere after the basic method is executed.
@AfterLast	The new code must be executed as the last one after the basic method is executed.

@Before(*Group*)	The new code must be executed before related new code in another group.
@After(*Group*)	The new code must be executed after related new code in another group.
@Replace	The new code totally replaces the basic method, without calling the super method.

Again similar to the algorithm in **OOD**, **AOP** and **FOP** in checking the consistence with the annotations, we first validate the compatibilities between the annotations. Then we build the execution order and during the building process we check the annotations one by one to see whether they are compatible with the execution order. Basically the errors are similar to the ones in **FOP** so we omit them here.

5 Implementation and Examples

5.1 OOD

We have finished the implementation for this part with the annotations. The implementation is against the solution at 4.1.

Here we give an example to show how the annotations are used. Suppose we have a base class as Figure 52 shows. This class requires that the subclass must override this method and at the same time, the subclass must call super at the first statement.

```
public class Sample {

    /**
     * This class has a set of annotations:
     *
     *  <ol><li>@Overrides(type=Type.MUST) -- Declares subclasses must
     *  override this method, or suffer an error.
     *  </ol>
     */
    @Overrides(type=Overrides.MUST)
    @Before(type=Before.MUST)
    public void doThis() {
        System.out.println ("SAMPLE");
    }
}
```

Figure 52. The base class

```
public class SampleSubclass extends Sample {

    /**
     * This class has a set of annotations:
     *
     *  <ol><li>@Overrides -- Declares that this subclass overrides the
     *  method in superclass.
     *  </ol>
     */
    @Overrides
    @After
    public void doThis() {

        System.out.println("DO MY STUFF");
        super.doThis();
    }
}
```

Figure 53. The subclass

Now we have a subclass which extends the base class, as Figure 53 shows. In this class the extended method annotates itself to call super at the last statement. Apparently, this conflicts with its parent class. So an error is reported on this:

```
Before:MUST incompatible with After
```

Though we used the algorithm stated in Section 4.1, we can use same algorithm as we used for **AOP** and **FOP** to find out the conflicts, by building the execution order. For example, in the above example we notice that the base class annotates that the method must be called at the first statement. If we build the execution order, we can easily find out that it will not be executed at the first statement. By this way, we don't even need to depend on the annotations in the subclass, because we directly use the actual code in the overriding method.

5.2 AOP

In this part, we show the implementation of the annotations and the checking engine. Then we use some examples to demonstrate how the annotations and the engine work.

- **Implementation of Annotations**

The latest JDK versions (from 5.0) support annotations, so that we can take advantage of this. An example implementation of an annotation is as Figure 54 shows. This example demonstrates how to define a relative before annotation. In this definition, note that the annotation type declaration is itself annotated. Such annotations are called meta-annotations. The first line, **@Retention (RetentionPolicy.RUNTIME)**, indicates that annotations with this type are to be kept by the compiler during runtime. The second **@Target (ElementType.METHOD)** indicates that this annotation type can be used to annotate only method declarations.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Before(
    String value();
)
```

Figure 54. Definition of relative **@Before** annotation

The use of this annotation can be demonstrated as Figure 55 shows. This annotation means that the advice must be executed before the advice in aspect B.

```
public aspect A{
   pointcut myF():    call(void F.f1());
   @Before("B")
   before(): myF() {
        // do something
   }
}
```

Figure 55. Example use of relative **@Before** annotation

- **Implementation of the Engine**

For the solution in Section 4.1, we can implement a checking engine for **AOP**. The implementation can find out all the warnings and errors. We have built enough testing cases to test our engine.

Major classes include:

Checker	The main class that does the checking. There is a check() can be called to find out the warnings

	and errors when the environment is built up.
Environment	Represent the whole environment, which contains all the classes, aspects, methods, advice, annotations, etc from the source files.
Class	Represent a class.
Method	Represent a method.
Aspect	Represent an aspect.
Advice	Represent a piece of advice.
Annotation	Represent an annotation.
JoinPoints	Represent the joinpoints for a piece of advice.
Pointcut	Represent a pointcut.
PrecedenceManager	Manage the precedence between the aspects.
Result	Represent the result generated after a checking is done.
Warning	The parent class for all the warnings.
Error	The parent class for all the errors.

The enumerations include:

AdviceType	The advice types, including BEFORE, AFTER and AROUND advice.
AnnotationType	The annotation types, including all the types listed in Section 4.1.
PrecedenceType	Represent the precedence types between aspects. This includes HIGHER, LOWER, INDEPENDENT, ITSELF and UNDIFINED.

A lot of testing cases (examples of the **AOP** files for all of the warning and errors) have been developed against the tool.

- **A small example**

Here is an example of finding out one of the warnings: annotation is missed.

Suppose the existing class and methods are as Figure 56 shows.

```
public class F {
  void f1() { }
}
```

Figure 56. Existing class

We have an aspect which tries to use a piece of advice before the `f1()` method, as Figure 57 shows.

```
public aspect A {
  pointcut myF():  call(void F.f1());
  before(): myF() {
     // do something
  }
}
```

Figure 57. An aspect which contains the warning

By using our engine to check the source code of the classes and aspects, the engine will report the following message:

WARNING: No annotation was found on advice: BEFORE myF() at line 4 at aspect A

- **Another small example**

We have another example for finding out the errors. Suppose we have two aspects on the above existing class, as Figure 58 and Figure 59 show.

```
public aspect A {
  pointcut myF():  call(void F.f1());
  @Before
  before(): myF() {
     // do something
  }
}
```

Figure 58. An aspect that actually executes first

```
public aspect B {
  pointcut myF():  call(void F.f1());
  @BeforeFirst
  before(): myF() {
     // do something
  }
}
```

Figure 59. An aspect that is declared to execute first but actually not

If we have a declared precedence statement on the two aspects: *declare precedence: A, B,* we can see aspect A would be executed before aspect B.

However, the annotation in aspect B means that the advice should be executed first before the method. So a conflict exists for the code. Our engine can find out this error by combining all the aspects and the precedence between them, and report the following error message:

```
ERROR: CONFLICTS FOUND: Advice BEFORE myF() in aspect B at line 4 was declared
to be the FIRST to be executed, but advice BEFORE myF() at line 4 in aspect
A can be executed before it
```

- **The TeleCom example**

Now let us use our solution against the TeleCom system we mentioned above. The programmer first finished the Connection class. Then the new requirement comes so two programmers start to work on it. Before they started to work, they came to an agreement that the timing should be calculated first before the billing. So same as above, a helper class `Timer` is created. The difference is the use of the annotation for the aspects. They created the two aspects as Figure 60 shows.

```
public aspect Timing{
    pointcut start (): call(void
Connection.start());
    @BeforeFirst
    before(): start () {
    Timer.start();
    }
    pointcut end (): call(void
     Connection.stop());
        @AfterFirst
    after(): end () {
    Timer.stop();
    }
}
```

```
public aspect Billing{
    private long rate = 10;
    pointcut end (): call(void
    Connection.stop());
    @AfterLast
    after():end () {
      long time = Timer.getTime();
     long cost = rate * time;
     System.out.println("The cost
     is: " + cost);
   }
}
```

Figure 60. Aspects with annotations

Again, they didn't specify the precedence between the two aspects, so the got the following results by our tool.

WARNING: No precedence declared between Aspect Timing and Aspect Billing, but they are dependent on each other.
ERROR: CONFLICTS FOUND: Advice AFTER end() in aspect Timing at line 9 was declared to be the FIRST to be executed, but advice AFTER end() at line 5 in aspect Billing can be executed before it

Then they use the declared statements to specify the precedence between the two aspects by "*declare precedence: Timing, Billing*". However, this is a wrong precedence. Our tool will find out this and again report the following error:

ERROR: CONFLICTS FOUND: Advice AFTER end() in aspect Timing at line 9 was declared to be the FIRST to be executed, but advice AFTER end() at line 5 in aspect Billing can be executed before it

Then they rethink about the precedence and find out they used the wrong precedence. With the correct precedence declared, our tool will report no errors.

From the examples, you can see that we do not need to wait till the runtime to find out the wrong semantics caused by the precedence issue. And not to say, some wrong semantics can not be found just in runtime by analyzing the output in a short time.

5.3 FOP / IOP

For the solution in Section 4.2, we can implement a checking engine for **FOP**. The implementation can find out all the errors. We have built enough testing cases to test our engine.

Major classes include:

Checker	The main class that does the checking.
Equation	Represent equation of the layers.
Layer	Represent a layer.
Class	Represent a class.
Method	Represent a normal method.
RefinedMethod	Represent a refined method.
MethodBody	Represent the method body for a method.
CodeFragment	Represent code fragment.
SuperCall	Represent a Super call. Subclass of CodeFragment.
Annotation	Represent an annotation.
RelativeAnnotation	Represent a relative annotation.
Concerns	Represent the concerns for one layer.
Result	Represent the result generated after a checking is done.
Error	The parent class for all the errors.

The enumerations include:

ExecutionAnnotationType	The annotation types, including all the types listed in Section 4.2.
PrecedenceType	Represent the precedence types between layers. This includes LEFT, RIGHT and ITSELF.

A lot of testing cases (examples of the **FOP** files for all of the errors) have been developed against the tool.

- **A Small Example**

This is an example about the concerns. Suppose we have a layer called layer1 which required a flowright concern "Bool deckBuilt". However, none of the layers on its left provides that concern. Our tool will report an error like this:

```
ERROR: Required flowright Bool deckBuilt at layer layer1 can not be satisfied.
```

If a new layer is added to the equation on the left and provides the deckBuilt flowright concern, the error will be eliminated.

- **Another Small Example**

This example is about the `Super` call invocation times. As we mentioned above, the `Super` call must be invocated for at most one time in a refined method.

If multiple invocations of `Super` happen as Figure 61 shows, an error would be reported:

```
ERROR: In Layer: Layer_1->Class:A->Method:int m(int), at line 4
      You must call Super exactly for one time, but you called 2 times.
```

```
refines class A{
  int m(int i) {
    Super().m(i);
    Super().m(i);
    // do something
  }
 }
```

Figure 61. Multiple `Super` calls are invocated

- **Third Example**

This example is about the annotations that are not compatible with their own code. For example, it will be wrong if a **@Before** annotation is used on a refined method which does not call `Super` at the last line. Figure 62 shows this.

```
refines class A{
   @Before
   int m(int i) {
     Super().m(i);
     // do something
   }
 }
```

Figure 62. Annotations incompatible with their own code

An error should be reported on this:

```
ERROR: In Layer: Layer_1->Class:A->Method:int m(int), at line 4
      The annotation BEFORE conflicts with its actual code.
```

- **Fourth Example**

This example is about the error when annotations are not compatible with the final woven code. Suppose a refined method is declared to be the first one executed before the basic method, however another refined method will be executed before it. Figure 63 shows the one declared to be executed first. Figure 64 shows the one is actually executed first. The method in Figure 64 is on the left layer of Figure 63.

```
refines class A{
   @BeforeFirst
   int m(int i) {
     // do something
     Super().m(i);
   }
 }
```

Figure 63. A method declared to be executed first

```
refines class A{
   int m(int i) {
     // do something
     Super().m(i);
   }
 }
```

Figure 64. A method that is actually executed first

An error is reported on this by our tool:

```
RROR: In Layer: BaseLayer->Class:A->Method:int m(int), at line 4
      It is declared to be executed PREFIRST, but the method in
      Layer:Layer_1->Class:A can be executed before it.
```

- **The TeleCom Example**

Now let us use our tool against the TeleCom example we showed above in the motivation part, to see whether our tool can solve the problem very well.

When working on the TeleCom example, the only difference is that the programmer needs to annotate the methods for *Timing* and *Billing*, as Figure 65 shows.

```
refines class Connection{
   @Before
    void start(){
        Timer.start();
        Super().start();
    }
    @Before("Billing")
    void stop(){
        Super().stop();
        Timer.stop();
    }
}
```

```
refines class Connection{
    private long rate = 10;
    @After("Timing")
    void stop(){
        Super().stop();
        long time = Timer.getTime();
        long cost = rate * time;
        System.out.println("The cost
is: " + cost);
    }
}
```

Figure 65. Timing (left) and Billing (right) with annotations

If *Timing* is on the right of *Billing* in the equation (which means *Timing* is the nearest layer to the base layer), the equation is expected and everything should be working okay. Our tool will not report any errors. However if *Billing* is on the right of the *Timing*, the equation would be wrong, because the time should be calculated before the bill. The annotations tell this too. So with our tool, an error is reported as below:

```
ERROR: In Layer: Timing->Class:Connection->Method:void stop(), at line 4
       It is declared to be executed BEFORE the method in Layer:Billing, but
       this is not guaranteed.
```

5.4 The Unified Model

From the analysis above on different design methodologies, we can have a general solution for the design problem on how to specify the designer intent. Figure 66 shows the architecture of such a solution. We can see that,

1. Annotations can be widely adopted to solve the problem for different design methodologies.
2. We need a specific parser for each design methodology. The parser is used to retrieve the annotation information and the other information from the source code.
3. A single back-end evaluation model can be adopted to unify all methodologies. This model constructs the directed multigraph, and evaluates the graph based on the semantics of the annotations as determined by the compatibility matrix (i.e. see section 4.1), together with the final execution order.

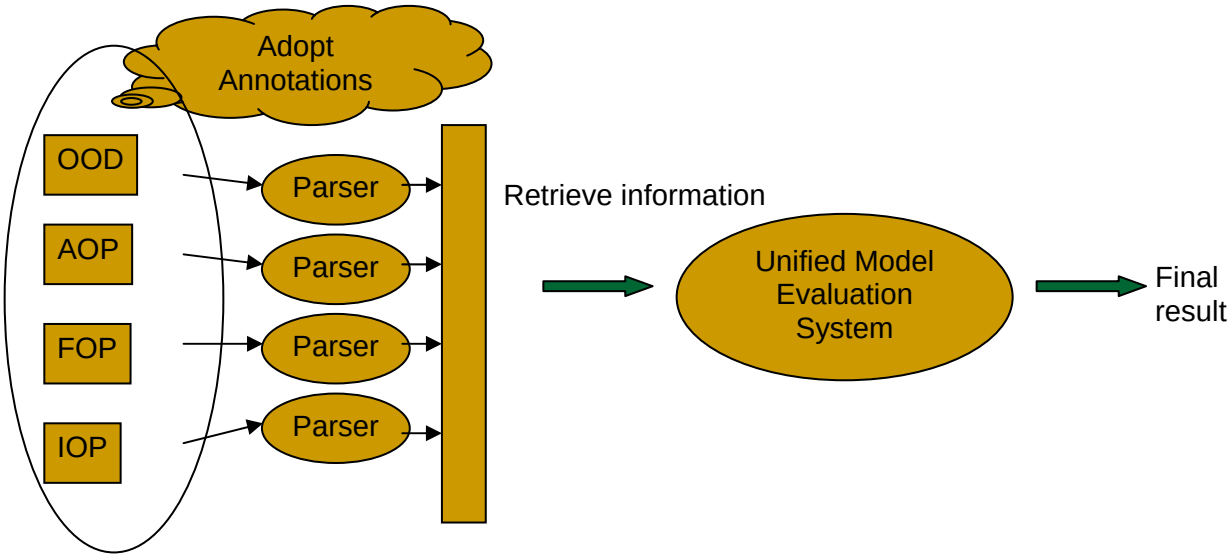

Figure 66 The general solution

In our implementation for the methodologies, we have implemented simple parsers to retrieve the information. For example, for **AOP**, we use class reflection to retrieve the annotations information and the class information. For **FOP**, we retrieve the information by parsing the source code line by line.While those parsers are still simple, they work fine for the given examples. More general parsers should be finished for different methodologies.

We have implemented evaluation systems for each methodology we investigated, but the unified

model is not given. This is also stated as future work. When designing this model in future, we should consider unifying the evaluation strategies for different methodology. For **OOD**, we adopted compatibility matrix and only consider the annotations compatibility, however for **AOP** and **FOP**, we built final execution order to identify the annotation conflicts. The two strategies should be put together to improve the evaluation.

6 Conclusion and Future Work

We believe see that our annotations can solve the precedence issue very well in **OOD**, **AOP** and **FOP**. The best reason to support the annotation idea is that we think designers should always know the relationship between the classes, aspects or layers, thus they should always know where the aspects or layers should be applied in whole system. Annotation enables the designers to control the semantics of the system. It is like when building a wall, the designer annotates every brick with a number, thus the workers know where to put the bricks in the wall.

Thus we can claim that we can guarantee that a correct semantics is adopted for the precedence issue.

For future work, as we already pointed out in the above section, more general parsers should be implemented for different design methodologies. We have implemented simple parsers for different methodology in our implementation. Besides it may be useful to use our research results with the existing IDEs such as Eclipse and AHEAD, for instance a plug-in in Eclipse may accelerate the use of the annotations. A unified model to evaluate the annotations should also be designed.

7 References

[1] D. Batory, J.N. Sarvela, and A. Rauschmayer, “Scaling Step-Wise Refinement”. IEEE TSE, June 2004.

[2] AHEAD Tool Suite. www.cs.utexas.edu/users/schwartz/ATS.html.

[3] ACDK, http://web.cs.wpi.edu/~heineman/html/research_/research.html.

[4] S. Apel and D. Batory. “When to Use Features and Aspects? A Case Study”, GPCE 2006.

[5] D. Batory, J. Liu, and J.N. Sarvela. “Refinements and Multi-Dimensional Separation of Concerns”, ACM SIGSOFT 2003.

[6] S. Chiba, “Program Transformation with Reflection and Aspect-Oriented Programming”, in Generative and Transformational Techniques in Software Engineering, LNCS 4143, 2006.

[7] E. Hilsdale and J. Hugunin. “Advice Weaving in AspectJ”, AOSD 2004.

[8] H. Rajan and K.J. Sullivan. “Classpects: Unifying Aspect- and Object-Oriented Programming”, ICSE 2005.

[9] D. Smith. “A Generative Approach to Aspect-Oriented Programming”, GPCE 2004.

[10] Christian Prehofer, “Semantic Reasoning about Feature Composition via multiple Aspect-weavings”, GPCE 2006.

[11] J. Liu, D. Batory, and C. Lengauer, "Feature Oriented Refactoring of Legacy Applications", International Conference on Software Engineering 2006, Shanghai, China.

[12] J. Liu, D. Batory, and S. Nedunuri, "Modeling Interactions in Feature Oriented Designs", International Conference on Feature Interactions (ICFI), June 2005.

[13] D. Batory, "Feature-Oriented Programming and the AHEAD Tool Suite", International Conference on Software Engineering (ICSE), Edinburgh, Scotland, 2004.

[14] AspectJ, version 1.5.2, http://www.eclipse.org/aspectj/.

[15] D. Ancona, G. Lagorio, and E. Zucca, “Jam - A Smooth Extension of Java with Mixins”, ECOOP 2000.

[16] H. Shinomi and T. Tamai, “Impact Analysis of Weaving in Aspect-Oriented Programming”, ICSM 2005.

[17] S. Apel, T. Leich, and G. Saake. “Aspectual Mixin Layers: Aspects and Features in Concert”, ICSE 2006.

[18] M. Stoerzer, R. Sterr and F. Forster, “Detecting Precedence-Related Advice Interference”,

Technical Report TR #0607, University of Passau, Computer Science Department, Passau, Germany, July 2006.

[19] G. Kiczales et al. “Aspect-Oriented Programming”, ECOOP 1997.

[20] E. V. Berard, “Essays on Object-Oriented Software Engineering”, Volum1, Prentice Hall, 1993.

[21] P. C. Jorgensen, “Software Testing A Craftsman’s Approach”, second edition, CRC Press, 2002.

[22] L. M. Bergmans, “Towards detection of semantic conflicts between crosscutting concerns”, Proceedings of workshop AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003, 2003.

[23] R. Douence, P. Fradet, and M. S¨udholt, “A framework for the detection and resolution of aspect interactions”, in GPCE’02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, pages 173–188, London, UK, 2002. Springer-Verlag.

[24] J. L. Pryor and C. Marcos, “Solving conflicts in Aspect-Oriented applications”, In Proceedings of 4th Argentine Symposium in Software Engineering, Buenos Aires, Argentina, September 2003.

[25] IEEE, “IEEE Standard Glossary of Software Engineering Terminology”, New York, 1983.

[26] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In SIGSOFT ’04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, pages 147–158, New York, NY, USA, 2004. ACM Press.

[27] T. Ishio, S. Kusomoto, and K. Inoue. Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on Software Maintenance, Washington, DC, USA, 2004. IEEE Computer Society.

[28] http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci212681,00.html

[29] Don Batory. “A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite”ftp://ftp.cs.utexas.edu/pub/predator/FOPTutorial.pdf

[30]Christian Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects”, ECOOP, 1997

[31] R. Lopez-Herrejon and D. Batory, “Taming Aspect Composition: A Functional Approach”.The University of Texas at Austin, Department of Computer Sciences. Technical

Report TR-05-27. June 8, 2005

[32]Sven Apel, Thomas Leich, Marko Rosenm¨uller, and Gunter Saake, “Combining FeatureOriented and AspectOriented Programming to Support Software Evolution”, ECOOP, 2005

[33] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[34] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck and M.-H. Durand, “Testing Levels for Object-Oriented Software”, ICSE 2000, Limerick, Ireland

[35] Frans Sanen, Eddy Truyen, Bart De Win, Wouter Joosen, Neil Loughran, Geoff Coulson, Awais Rashid, Andronikos Nedos, Andrew Jackson, Siobhan Clarke, “Study on interaction issues”, Katholieke Universiteit Leuven, Leuven, AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, 28 February 2006, pp 1-31

[36] Gregor Kiczales c, “Getting started with ASPECTJ “, Communications of the ACM, Volume 44 , Issue 10 (October 2001), Pages: 59 - 65

[37] Sergei Kojarski, David H. Lorenz, “Identifying Feature Interactions in Multi-Language Aspect-Oriented Frameworks”, Minneapolis, MN, 29th International Conference on Software Engineering (ICSE'07) pp. 147-157