ENVIRONMENT GENERATION TOOL FOR ENABLING ASPECT VERIFICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ŞENOL LOKMAN ALDANMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

JUNE 2010

Approval of the Graduate School of Informatics

_____

Prof. Dr. Nazife Baykal

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Assist. Prof. Dr. Tuğba T. Temizel

Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Assist. Prof. Dr. Aysu Betin Can

Supervisor

Examining Committee Members

Prof. Dr. Semih Bilgen (METU, EEE)_____

Assist. Prof. Dr. Aysu B. Can (METU, II)_____

Assist. Prof. Dr. Erhan Eren (METU, II)_____

Assist. Prof. Dr. Altan Koçyiğit (METU, II)_____

Assoc. Prof. Dr. Halit Oğuztüzün (METU, CENG)_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Şenol Lokman Aldanmaz

Signature : _____

ABSTRACT

ENVIRONMENT GENERATION TOOL
FOR ENABLING ASPECT VERIFICATION

Aldanmaz, Şenol Lokman
M.S., Department of Information Systems
Supervisor: Assist. Prof. Dr. Aysu Betin Can

June 2010, 96 pages

Aspects are units of aspect oriented programming developed for influencing the software behavior. In order to use an aspect confidently in any software, first it should be verified. For verification of an aspect, the mock classes for the original software should be prepared. These mock classes are a model of the aspect environment which the aspect is woven. In this study, considering that there are not enough tools for supporting the aspect oriented programming developers, we have developed a tool for enabling aspect verification and unit testing. The tool enables verification by generating the general environment of the aspect. By this tool the users are ensured to focus on the verification of aspects isolated from woven software.

Keywords: Aspect Oriented Programming, Enabling Software Verification, Code Generation, AspectJ

ÖZ

İLGİ DOĞRULAMASINA OLANAK SAĞLAYAN
ORTAM TÜRETİCİ ARAÇ

Aldanmaz, Şenol Lokman
Yüksek Lisans, Bilişim Sistemleri Bölümü
Tez Yöneticisi: Yrd. Doç. Dr. Aysu Betin Can

Haziran 2010, 96 sayfa

İlgiler, ilgi yönelimli programlamada yazılım davranışını etkilemek için geliştirilen birimlerdir. Bir ilgiyi herhangi bir yazılımda güvenli olarak kullanabilmek için ilk önce o ilgi doğrulanmalıdır. Bir ilginin doğrulanması için asıl yazılımın taklit sınıfları hazırlanmalıdır. Bu taklit sınıflar ilgilinin örülü olduğu ilgi ortamının bir modelidir. Biz bu çalışmada, ilgi yönelimli programlama geliştiricilerini destekleyecek yeterli miktarda araç olmadığını göz önünde bulundurarak, ilginin doğrulanmasına ve birim testine olanak sağlayan bir araç geliştirdik. Geliştirilen araç ilginin genel ortamını üreterek doğrulamaya olanak sağlar. Bu araçla kullanıcılar, örülü olduğu yazılımdan yalıtılmış olan ilgilerin doğrulanmasına odaklanmaları konusunda temin edilmektedirler.

Anahtar Kelimeler: İlgi Yönelimli Programlama, Yazılım Doğrulamasına Olanak Sağlamak, Kod Üretimi, AspectJ

To my parents, brother and tatiş

ACKNOWLEDGEMENTS

I would like to thank my advisor Assist. Prof. Dr. Aysu Betin Can for her guidance and support with sympathy and patience through out my study.

Also, I thank my lovely fiance and family for their endless support and motivation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS / ACRONYMS

JavaCC	: Java Compiler Compiler
JML	: Java Modeling Language
JPF	: Java Path Finder
JTB	: Java Tree Builder
UML	: Unified Modeling Language

CHAPTER I

INTRODUCTION

Aspect oriented programming is first proposed nearly a decade ago and it enhances its popularity as the time passes. It is expected to be a significant step in the growth of software world. Instead of being an alternative for object oriented programming, aspect oriented programming is a different view and a complementary methodology of it.

Since aspect oriented programming is a relatively fresh work area in software world (became available in 2001) there are incomplete parts of it and one of those missing parts, verification of aspect oriented programs, is studied by several people from different perspectives. As any other type of programs, aspect oriented programs need to be verified within a software verification process and they should be unit tested. Considering the verification and unit testing processes of other programming methodologies, aspect oriented programming developers need to be supported by several automatic operations during these processes.

Our motivation in this study is to support developers of aspect oriented programming by enabling verification and unit testing. We enable verification and unit testing of aspect oriented programs by isolating aspects and automatically generating its general environment.

I.1 Aspect Oriented Programming Methodology

Aspect oriented programming methodology is not a totally different or alternative methodology among programming methodologies. Actually, it can be considered as a supporting methodology for object oriented programming. In object oriented programming highly cohesive coding is preferred always, however sometimes there appears leakages of that methodology in terms of cohesion and there is no way to handle it with it's own elements. Aspect oriented programming comes to the help of object oriented programming at this point. As an example, consider the logging feature and its implementation in object oriented programming. The logging task needs to be performed with a method call in each procedure of the system; hence, every method in every class need to do an additional work, which is not related with it directly and this situation damages the high cohesion goal. Aspect oriented programming exist for handling such cross-cutting concerns. It is possible to do such a general logging task without changing the original code in aspect oriented programming by using aspects [1]. All of the processes related with logging task are kept in a logging aspect. Nothing appears in the original code related with logging task. Logging aspect assumes the control at specific points that it determines and does the processes related with logging task. It is completely beyond the control of original code. Thus, the original code can focus on its own processes [2]. Details about aspect oriented programming are discussed in the next chapter.

I.2 Software Verification and Unit Testing

The goal of software verification is to examine the correctness of software. Until last two decades, the common practice to verify code was the expert people to model that code at high level and those models were subject to verification. Since modeling is not a cheap process, recently tendency originated through verification of the code directly, instead of model verification, however it is not so easy to check the correctness of the code directly [3].

Unit testing is one of the software verification and testing techniques that focus on testing the smallest testable units of a program. It aims to make tests on isolated parts of software to find the actual source of problem more easily [4].

We aim to prepare the environment of an aspect for unit testing and for making it possible to verify with an automated verification tool, such as Java PathFinder (JPF) [5]. Details including software verification, JPF tool and unit testing are discussed in the next chapter.

I.3 Scope of This Thesis

In this thesis, we aim to feather the aspect oriented program developers' nest on the verification of the aspects. By providing a tool that automatically isolates an aspect we support the developers during their verification process.

As any other software program, aspect oriented programs need to be verified to have a reliable program at the end. There are several ways to verify an aspect oriented program; however, in this study, we focus on the verification of aspects, without getting the base program from the developer. Actually we aim to prepare a temporary base program from the given aspect without any developer effort.

The method that we suggest for verification of aspects is as follows. Given an aspect, generate its environment that enables the aspect to show all of its possible behaviors. Thus aspect oriented program developer can do verification of aspects with the generated base code easily. The advantage of using the generated base code is finding the problematic points in only the aspects and focusing only on the aspect code problems. If the developer does not use this tool or the code generated by this tool, there are two choices for him. First one is using the original base code. Disadvantage of this choice is that during verification problems may be both the problems of the aspect code and the problems of base code. This means that the developer cannot focus on the problems of aspects only. The other choice is writing a

general environment for focusing only on the aspect problems, however again at this time it is not possible to guarantee that the written environment is errorless.

Basically, the subject of this thesis is parsing the aspect code and generating base code from the information that is collected during parsing. The generated base code can be thought as a mock class of the real base program to verify the aspect code correctly. Since aspect oriented programming has different implementations for different programming languages, it is not possible to generalize our project on all of the programming languages. We focus on AspectJ programming language, which represents aspect oriented programming that is implemented for Java programming language [6]. It is one of the most popular aspect oriented programming languages.

I.4 Outline of This Thesis

This document consists of seven chapters to describe the work done in this thesis. The first chapter is the introductory part of the document. In this chapter, to give an initial idea about the topic, the necessary subjects and their brief descriptions are identified.

Chapter II presents the background information of this study and related work on verification of aspect oriented programming. The necessary information, to fully understand this thesis study, is provided in this chapter.

In chapter III, the need for such a study on verification of aspects and the problems related with verification of aspect oriented programming are explained.

Chapter IV is reserved for description of the general architecture of our solution to the problem. Several parts of our solution are displayed at a high level in this chapter of the document.

Chapter V aims to explain the method that we suggest for enabling verification of aspects in aspect oriented programming. Details of several parts of our solution are explained in this chapter with their logics.

In chapter VI several examples of aspect code are experimented. Possible tricky usages of AspectJ code that is supported by our tool is displayed at this chapter.

Last chapter presents the conclusion of this thesis study and the future work. This study is expected to present a useful tool for AspectJ software developers.

CHAPTER II

BACKGROUND AND RELATED WORK

Aspect oriented programming, AspectJ programming language and its elements constitute the first part of this chapter. To realize the work done in this study, it is necessary to have an idea about the terms and usages in aspect oriented programming. The summarized information is sufficient for understanding the remaining chapters of the document.

Software verification and unit testing form the second part of the chapter. Instead of helping the realization of the terms and the idea in the thesis, this part helps to understand the purpose of this thesis study, because the study aims to help verification of aspect oriented programs.

The third part is about JTB. JTB is an open source application that is used for facilitating the parsing process of aspects in AspectJ [7]. Only the necessary parts related with this thesis study is explained in this document.

After the background information, related studies on the topic are discussed at last part of this chapter.

II.1 Aspect Oriented Programming

Object oriented programming is the dominating software development methodology for a long time, because it is powerful from several points of views. It is possible to prepare lowly coupled, highly cohesive software designs in object oriented programming and this makes it more popular [8]. However, nothing is perfect, and of course there are points that object oriented programming methodology cannot fulfill perfectly.

As an example, sometimes developers need to do the same thing and write the same code again and again for every class or for every method. Logging is a suitable example for here. In an object oriented programming language the developer needs to write the code block for logging in all methods. This is an unnecessary extra load and there is no way to handle such an extra work easily in object oriented programming. Aspect oriented programming is in the service of the developer in such a case.

Aspect oriented programming is a developing technology aiming to enhance cohesion where object oriented programming fails short to do it. In order to achieve this goal, it adopts separation of concern principle [9]. As mentioned above for the logging example, instead of interfering all of the classes, a single aspect handles the logging process and localizes the code to a shared place. A keyword here is cross-cutting concern, in other words aspects crosscut classes. In aspect oriented programming those classes are called as base classes and through this document this definition is used frequently [9]. Similarly, variables of base classes are referred as base variables and methods of base classes are referred as base methods. To sum up, in addition to classical object oriented programming elements, there are aspects in aspect oriented programming and these aspects are woven into the classes. After this process the behavior of the resultant program involves both the behavior of the base program and the behavior of the cross-cutting concern (aspect) [10].

There are several programming languages that are developed for aspect oriented programming. For this thesis study we have chosen AspectJ which is an aspect oriented programming language based on the Java programming language. Aspects in AspectJ can be considered like the classes in object oriented programming although there are differences. A basic difference, for example is that it is not possible to instantiate an aspect on its own like a class. The aspects have to depend on classes and they do not exist when there is no class [11]. The elements that may exist in the content of an aspect can be grouped under three sub-topics as point-cuts, advices and other elements [12].

To explain these concepts, it is better to work with an example. Although the definitions of concepts like point-cut, advice, intertype variable and so on are not provided yet, by this example a general idea is given about them. The definitions of AspectJ specific concepts are given in the following sections. A class in Figure 1 and an aspect in Figure 2 that is interfering it are presented and aspect oriented programming concepts are going to be analyzed on this code blocks.

```
public class Student {                                          //line 1
        private float paidMoney = 0;                            //line 2
        private boolean isRegistered = false;                   //line 3

        public void payMoney(float money) {                     //line 4
                paidMoney = money;                              //line 5
        }                                                       //line 6
        public void register() {                                //line 7
                isRegistered = true;                            //line 8
        }                                                       //line 9

        public boolean isStudentRegistered() {                  //line 10
                return isRegistered;                            //line 11
        }                                                       //line 12
}                                                               //line 13
```

Figure 1: Student Class

```
public aspect StudentController {                                    //line 1
    private boolean Student.isPaidMoney = false;                     //line 2

    public pointcut payMoneyPct(float money) :                       //line 3
        call(public void Student.payMoney(float)) && args(money);    //line 4

    //Advice 1
    /* @ ensures                                                     //line 5
     @ isPaidMoney == true;                                          //line 6
     */                                                              //line 7
     after(float money) : payMoneyPct(money) {                       //line 8
            isPaidMoney = true;                                      //line 9
    }                                                                //line 10

    //Advice 2
     before() : set(public boolean Student.isRegistered) {           //line 11
            if(!isPaidMoney) {                                       //line 12
                System.out.println("Before registration              //line 13
                                    you should pay money!");         //line 14
            }                                                        //line 15
    }                                                                //line 16

    //Advice 3
    /* @ ensures                                                     //line 17
     @ (isPaidMoney == false) ==> student.isRegistered == false;     //line 18
    */                                                               //line 19
    after(Student student) :                                         //line 20
        call(public void Student.register()) && this(student) {      //line 21
            if(!isPaidMoney) {                                       //line 22
                isRegistered = false;                                //line 23
     }                                                               //line 24
    }                                                                //line 25

    //Advice 4
     void around() : get(public boolean Student.isRegistered) {      //line 26
            System.out.println("isRegister variable is read.");      //line 27
    }                                                                //line 28
}                                                                    //line 29
```

Figure 2: Student Controller Aspect

As presented in Figure 1, there is a class as `Student` with `paidMoney`, `isRegistered` attributes and `payMoney`, `register` methods. The methods aim to change the values of attributes.

Assume that a university announced some restrictions on student registration process and one of them is “Nobody can register before paying the registration fee” There exist several methods to achieve this goal. For example, if a new variable as `isPaidMoney` is added into the `Student` class and is used as a flag for controlling

money payment the problem can be solved. Another solution may be controlling the amount of money. If `paidMoney` attribute equals to zero then registration would not be accepted. These are the methods that anybody can think at first sight. However in order to explain the concepts in AspectJ, the problem is solved with a different methodology.

The solution that AspectJ proposes is adding an aspect and leaving the code of `Student` class as is. The name of the aspect is chosen as `StudentController` and the content of it is given in Figure 2. Firstly, an intertype variable is declared (line 2) as `isPaidMoney` and its default value is set to false. This variable acts as if it is an attribute of the `Student` class (that is why it is called intertype variable).Then, a point-cut is defined as `payMoneyPct` (line 3-4), which is activated when `payMoney` method of `Student` class is called. Line 5-7 is corresponding to JML code that is telling the variable `isPaidMoney` is surely set to true when this advice (line 8-10) is processed. JML code is generally used for verification and specification purposes; however, in our study they are used for explanation like comment lines. Therefore we are not going to deal with JML and no detailed information is provided related with it. Line 8-10 declares an after advice, which is processed when the `payMoneyPct` is activated. Therefore, when the call to `payMoney` method of `Student` class is ended, `Advice 1` is triggered and then the advice sets the intertype variable `isPaidMoney` as true. The second advice (line 11-16) is activated before an assignment to the variable, `isRegistered` in `Student` class, is done. It displays a message by checking the variable `isPaidMoney`, if money is not paid then a message declaring that registration is prohibited is shown. Line 17-19 constitutes JML code about `Advice 3`. It explains that if money is not paid, then registration is not accepted. This is done by the code of lines 20-25.

II.1.1 Point-cuts

Point-cuts are the most differentiating parts of aspect oriented programs comparing with other programming methodologies. Before explaining point-cuts, it is necessary to have an idea about the definition of join points. Join points can be defined as the critical and well-defined points through the execution of a program. Actually, they are points that aspect can join base program execution. Point-cuts can be defined as the elements, which are catching the join points during flow of a program. After join points are caught by point-cuts, advices affect the flow of program by using these point-cuts.

Below table is showing the basic and significant matches of join points with point-cuts [12].

Table 1: Join Points and Point-cuts [12]

	Join Point	**Point-cut**
1.	Method call	call(Method signature)
2.	Method execution	execution(Method signature)
3.	Constructor call	call(Constructor signature)
4.	Constructor execution	execution(Constructor signature)
5.	Field read	get(Field signature)
6.	Field write	set(Field signature)
7.	Object initialization	initialization(Constructor signature)
8.	Static initialization	staticinitialization(Type signature)

- The first join point is representing method calls. A simple example is shown in Figure 3 and Figure 4 from `StudentController` aspect.

Aspect code

Figure 3: Point-cut (call type)

Base code

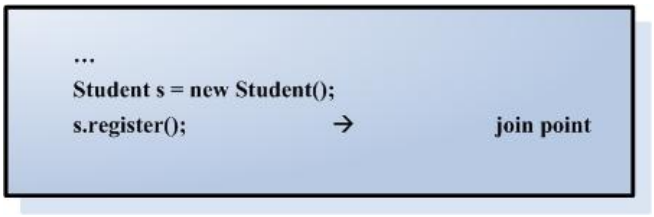


Figure 4: Join Point (call type)

- Second one is for execution of method instead of calling. For the same point-cut an example for execution is represented in Figure 5 and Figure 6.

  Aspect code

Figure 5: Point-cut (execution type)

Base code

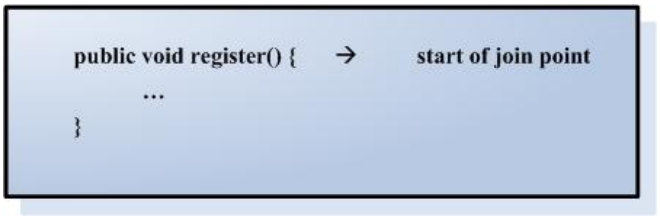


Figure 6: Join Point (execution type)

- Third join point is representing constructor calls. It is very similar to method calls.

  Aspect code

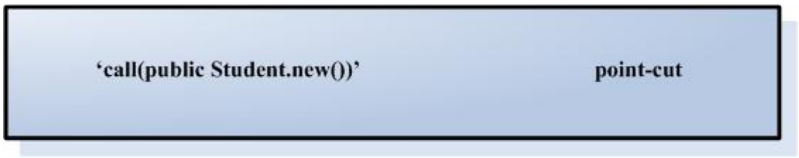


Figure 7: Point-cut (constructor call type)

  Base code

Figure 8: Join Point (constructor call type)

- Forth point-cut is for execution of constructor. For the same point-cut an example for execution is shown in Figure 9 and Figure 10.

  Aspect code

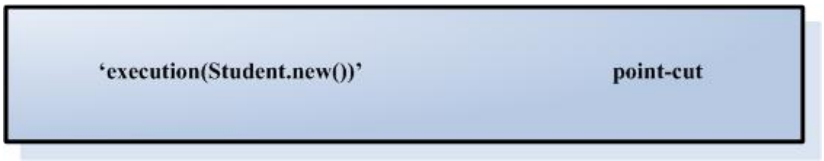


Figure 9: Point-cut (constructor execution type)

  Base code

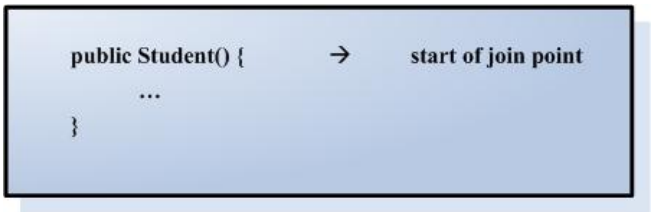


Figure 10: Join Point (constructor execution type)

- Fifth join point represents field reading process. A simple example is shown in Figure 11 and Figure 12 from `StudentController` aspect.

  Aspect code

Figure 11: Point-cut (get type)

  Base code

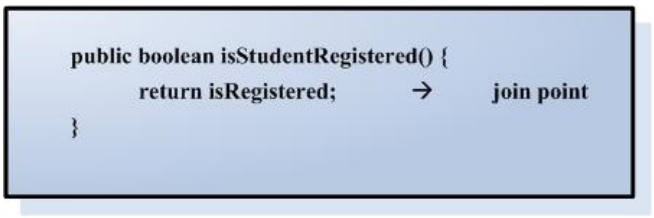


Figure 12: Join Point (get type)

- Join point six represents field writing process. A simple example is shown in Figure 13 and Figure 14 from `StudentController` aspect.

  Aspect code

Figure 13: Point-cut (set type)

  Base code

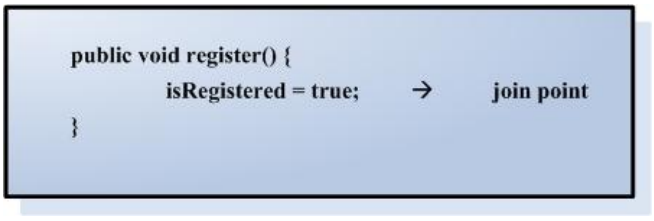


Figure 14: Join Point (set type)

Point-cuts are used in two different ways in AspectJ [12].

- Named point-cuts are the first type of that usage. Named point-cuts are defined as separate elements in an aspect and used in an advice with its name. Syntax of named point-cut is shown in Figure 15.

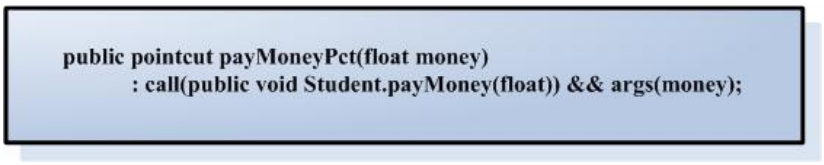


Figure 15: Named Point-cut

First element is an optional access sign `public`. Then a fixed expression comes, `pointcut`. After that, name of the point-cut is located, `payMoneyPct`. Then, the arguments of point-cut are located, but they are optional. After the arguments, the `':'` is used as a delimiter. The expressions after `':'` are changing according to the type of point-cut which is discussed above. The expression `&& args(money)` is used for getting and using the only argument of `payMoney` method inside the aspect code.

- Anonymous point-cuts are the second type of point-cut usage. They do not have a name and they are directly used in advices. Since they do not have names, it is impossible to use an anonymous point-cut more than once. An example of this usage is shown in Figure 16.

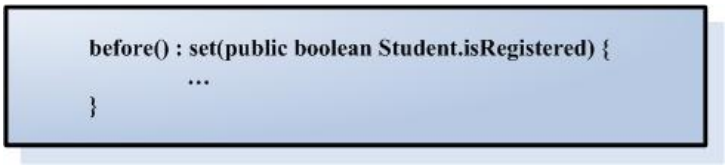


Figure 16: Anonymous Point-cut

`set(public boolean Student.isRegistered)` is point-cut part of the advice.

II.1.2 Advices

Point-cuts are important; however they are not useful without advices. Flow of program execution is captured by point-cuts and advice interfere that flow by its actions. The syntactical structure of an advice consist of three basic parts, these parts and their positions are shown in Figure 17 with an after advice example.

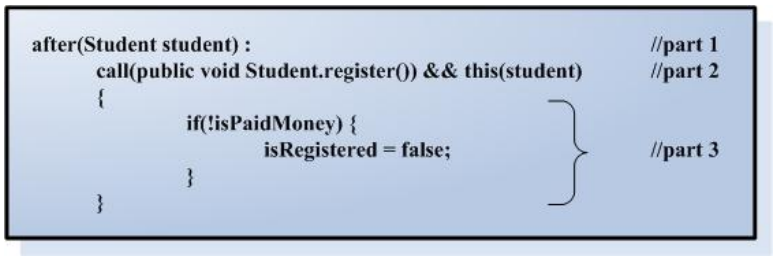


Figure 17: Sample Advice

Part 1 is the advice declaration. This part involves an advice identifier, which is `after` keyword this time, and optional advice parameters. These parameters are used to catch and use the original instances or values of variables in base code.

Part 2 represents the point-cut part of the advice and point-cuts are already discussed in the former section.

Third part of advice is called as advice body. Between the curly brackets any AspectJ code can be written and that code constitutes the body of advice [12].

There are three types of advices. Those types and their explanations are listed below [12].

- Before advice

  The body of this type of advice is executed before the execution of join point of the point-cut, which is second part of the advice. Normally, if no exception is thrown from the body of the advice, after the execution of advice body, the program execution flow continues from the join point. However, if an

exception is thrown during advice body execution, then the join point in the base code is not executed. A sample code of before advice is shown in Figure 18.

```
before() : set(public boolean Student.isRegistered) {
        …
}
```

Figure 18: Sample Before Advice

- After advice

  In this type of advice, advice body is executed after the execution of join point of point-cut. An example of after advice usage is shown in Figure 19.

```
after(float money) : payMoneyPct(money) {
        isPaidMoney = true;
}
```

Figure 19: Sample After Advice

- Around advice

  Around advice is a more talented advice compared to others. By using around advice, it is possible to take an action before and after a join point, ignore the join point, call the join point more than once or call it with different parameters. In an around advice body, a `proceed()` statement is used for the base code execution to continue from the join point. In Figure 20 there is an example of around advice usage.

```
void around() : get(public boolean Student.isRegistered) {
        System.out.println("isRegister variable is read.");
}
```

Figure 20: Sample Around Advice

II.1.3 Other Elements

In addition to the effects of point-cuts and advices to base program at run-time, there are some other elements in AspectJ to affect the compile-time of base program and behavior of these elements is called as static-crosscutting. It is possible to group static-crosscutting methods under four topics. They are member introduction, type hierarchy modification, compile-time error and warning declaration, and exception softening [12]. For this study it is sufficient to explain only the first element of these 4 elements.

A variable can be declared in an aspect and the type of that variable depends on its declaration type. There are two types of declaration of members in aspects; aspect member, intertype member. Aspect members are similar to attributes of a class, i.e. they belong to the aspect. On the other hand, intertype variables belong to the base class.

The difference of an intertype variable from an aspect variable is, there exist an owner class of an intertype variable and the variable is considered as a member of that class. Example in Figure 21 displays an intertype variable usage:

```
public aspect StudentController {
        private boolean Student.isPaidMoney = false;
          ...
}
```

Figure 21: Student Controller Aspect with Intertype Variable

`isPaidMoney` is a `boolean` type intertype variable with initial value `false`. This variable is declared as a member of `Student` class.

II.1.4 Weaving and Compilation Process

To compile the base code and aspects together AspectJ uses a different compiler on its own. It has also its own byte code weaver (ajc) just like its own compiler. AspectJ compiler checks if aspect code and base code are in coordination and they can run together. The process of this running in coordination is called as aspect weaving [12]. In other words, aspect weaving can be described as activating the advice code in aspects at their related join points.

Working strategy of AspectJ compiler at join points can be considered as method calls. Bodies of advices are weaved like static or final methods in Java code. Since they are like static method calls there is not much performance loss at run-time of AspectJ code [6].

II.2 Software Verification and Unit Testing

Automated software verification aims to find deadlocks, unspecified receptions, non-executable code, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes of the given input code [13]. Since it is yielding promising results, its usage area is expanding continuously.

One of the automated verification techniques is model checking. It is a very successful and popular technique in recent years. The model of the software should be prepared before the usage of model checking techniques and then verification process should run over that software model. Since the conversion process from software to a high level model is not an easy process, lately the idea of direct verification of implementation level code became popular [14]. Although our project is not directly related with software verification, the actual aim of this study is to generate environment for aspects and make it possible to verify aspect code with the generated environment.

Java Pathfinder [5] (JPF) is one of the most popular software verification tools. It is used for Java programming language. The aim of usage of JPF is described as focusing on the defects, coverage and several more information about the given input Java code.

Unit testing is one of the favorite testing methodologies to focus on small pieces of codes. It is a handy and easy way to obtain an accurate and quality program at the end [15]. The point that unit testing stands out with is isolation. Unit test of a code piece deals only with that code piece. Success of most of the huge software projects relies on the strict disciplined testing especially unit testing policy.

In unit testing to make an isolated test on small units of software a temporary environment should be prepared for that unit [15]. This is done via preparing the stubs or mocks of the classes that are in interaction with the unit. By this way, it is possible to focus on the problems of the unit independently from its environment.

II.3 JTB

Java Tree Builder (JTB) is a syntax tree builder application. It is also called as a frontend for the Java Compiler Compiler (JavaCC) parser generator [7]. JTB also provides visitor for traversing the syntax tree that it has built. Input and outputs of JTB are shown in Figure 22 [16].

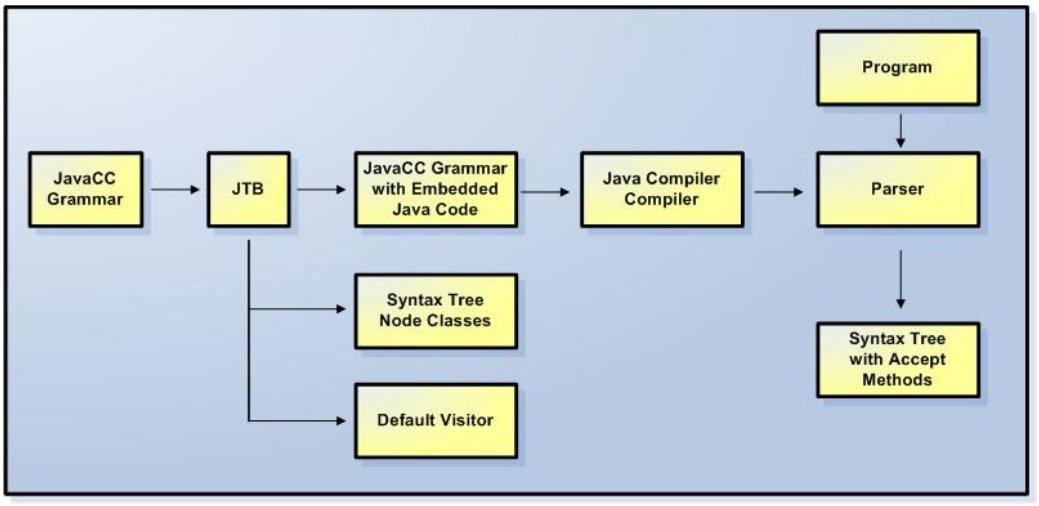


Figure 22: JTB input and outputs [7]

Input:

- JavaCC Grammar

In order JTB to be able to build a syntax tree, it is necessary to supply a JavaCC grammar. Every detail should be provided in this grammar file. Figure 23 shows a sample block from the content of our input grammar file.

```
void TypeDeclaration() :
{}
{
    LOOKAHEAD( ( "abstract" | "final" | "public" )* "class" )
    ClassDeclaration()
  |
    LOOKAHEAD( ( "abstract" | "final" | "public" )* "aspect" )
    AspectDeclaration()
  |
    InterfaceDeclaration()
  |
    ";"
}

void ClassDeclaration() :
{}
{
    (“abstract" | "final" | "public" )*
    UnmodifiedClassDeclaration()
}
```

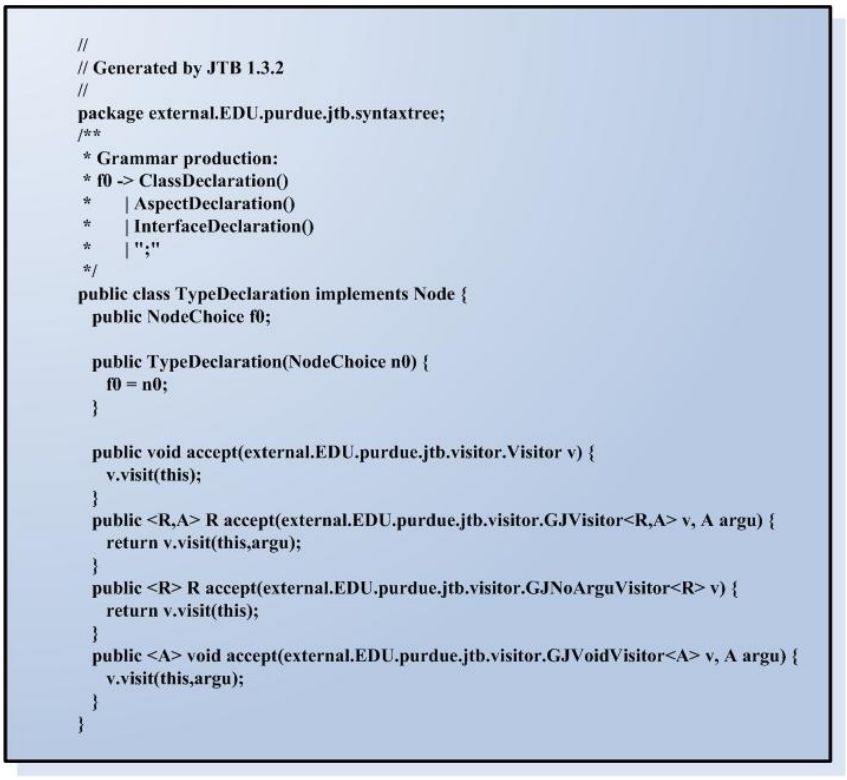
Figure 23: JavaCC Grammar Declarations

Here, `TypeDeclaration` syntax is a member of JavaCC grammar. Options about what kinds of syntaxes may correspond to a `TypeDeclaration` are declared. One of those options involves `ClassDeclaration` syntax and the syntax of `ClassDeclaration` is provided in above sample block as an insider member of `TypeDeclaration`. In the block above, declarations of `AspectDeclaration`, `InterfaceDeclaration`, and `UnmodifiedClassDeclaration` are not presented, however in the original grammar file of this study, hundreds of other declarations including these three declarations are supplied to JTB [7]. Most of these declarations are already provided in sample JavaCC grammar of JTB. The necessary declarations of AspectJ are added to this sample grammar file in this study and those declarations are explained Chapter V of this document. During the document the parser that we have generated by using JTB is referred as JTB parser.

Outputs:

- Syntax Tree Node Classes

For each declaration in JavaCC grammar file, a Java class is produced by JTB to make those elements open to visiting by visitor. For example, the class generated for `TypeDeclaration` is shown in Figure 24.



```
//
// Generated by JTB 1.3.2
//
package external.EDU.purdue.jtb.syntaxtree;
/**
 * Grammar production:
 * f0 -> ClassDeclaration()
 *       | AspectDeclaration()
 *       | InterfaceDeclaration()
 *       | ";"
 */
public class TypeDeclaration implements Node {
  public NodeChoice f0;

  public TypeDeclaration(NodeChoice n0) {
    f0 = n0;
  }

  public void accept(external.EDU.purdue.jtb.visitor.Visitor v) {
    v.visit(this);
  }
  public <R,A> R accept(external.EDU.purdue.jtb.visitor.GJVisitor<R,A> v, A argu) {
    return v.visit(this,argu);
  }
  public <R> R accept(external.EDU.purdue.jtb.visitor.GJNoArguVisitor<R> v) {
    return v.visit(this);
  }
  public <A> void accept(external.EDU.purdue.jtb.visitor.GJVoidVisitor<A> v, A argu) {
    v.visit(this,argu);
  }
}
```

Figure 24: Syntax Tree Node Classes

- Visitor Class

For visiting the nodes of generated syntax tree, a default visitor class is generated by JTB. The traversal of syntax tree methodology is depth-first visiting. In order for collecting information from the syntax tree, a class that is extending this default visitor class is necessary. For example an `AspectDepthFirstVisitor` that is extending the default visitor class should be prepared to capture the elements of aspects. Point-cuts, advices, intertype variables and so on, can be captured by this specialized visitor and then this information can be used in generation of the base classes for verification or mocks for unit testing.

- JavaCC Grammar with Embedded Java Code

JavaCC grammar is the third output of JTB. This grammar is generated with embedded Java code for building syntax tree. The visitor is traversing this tree instance.

II.4 Verification of Aspect Oriented Programs

In this section of the chapter, previous and ongoing studies on verification of aspect oriented programs are discussed. There exist several studies on the same and similar topics but the ideas and their views to handle those problems are different from our solution. In spite of the differences, it is beneficial to mention the related studies on a summary view before explaining the details of our study.

Larsson and Alexandersson studied verification of the aspects that add fault tolerance to software [14]. Although they focus on only the fault tolerance aspects, they also discuss general verification of aspect-oriented programs and compare several alternatives of verification methodologies of aspect-oriented programming. While discussing several alternatives they mainly introduce the idea about verifying the aspect without any base program. They propose that all kinds of point-cuts can be converted to an "around" advice in a way. Thus, if it is possible to handle the situation for "around" advice, then obtaining the final solution of the problem becomes easier. First, the advice body is split into two parts as before the "proceed" keyword and after the "proceed" keyword. Since there is not any dependency to return value of the "proceed" only the second part is considered for verification. After the "proceed" statement if there is a dependency to the return value of that statement then according to the dependency, the verification is branched out at that point. Therefore all possibilities are covered with this methodology. One of the advantages of this solution is lost of the dependency of aspect to the base program considering the verification process and this property is similar to the approach in our study. On the other hand, unlike we did, they do not try to handle all types of advices and eliminate others by converting them to "around" advice. Another advantage of choosing this approach is that it is possible to create libraries of aspects that involve dependable and reusable aspects.

In [17] a language for AspectJ, which is named as Pipa, is introduced and by using Pipa the transformation of AspectJ program into Java program with JML

specifications is explained. Aspect specifications are considered at transformation and JML is used for this purpose. Their proposal is modifying Aspectj compiler to handle the modification. Since conventional verifiers can handle the verification of Java and JML, it is possible to use those verification applications at this alternative. This approach serves the same purpose with our study although their methodology is completely different from ours.

Clifton and Leavens propose a different approach in [8]. There exists base code just like in [17] but this time, instead of converting to Java code, the constructors of AspectJ language is used as they are. There is need for too much analysis of the base code at this approach. Considering a point-cut of an advice in an aspect, whole base code should be traversed for each join point of that point-cut. Traverse is necessary, because while thinking from the verification view, for each join point a proof branch should be started. Additionally, extra new definitions are added that are coming from the language constructors of AspectJ language. At this approach base program and aspect code are separated. Unlike our study, for such an approach usage of base code as part of the proof is important and this major difference is the striking complexity and disadvantage of the related alternative.

Another methodology for verification in aspect oriented programming is proposed in [18]. This study aims to provide a solution for aspect verification by analyzing aspects modularly according to stable point-cut designators. The most critical problematic point of this study is its lack of support on most of AspectJ features. Instead of providing a much more implementation like ours, they aim to present their methodology with a basic implementation support. They also divide the aspect into modules and handle each module separately on the contrary to our approach.

Mostefaoui and Vachon propose another approach as verifying models of aspects that are written Aspect-UML language [19]. The most appearing disadvantage of this approach is the conversion effort from aspect to its model. Dissimilar to our study, they do not aim to enable verification in implementation level and they add a major conversion module to their methodology.

Although the aim of approach that is described in [20] is not directly related with enabling the verification of aspects, the used methodology makes it easy to verify aspects. In the approach aspects are transformed into alternating transition systems by using several properties. By using those properties it is easier to verify aspect code. This approach includes a conversion process unlike our approach and it is similar to converting the aspects into models.

[21] and [22] explain a tool that is developed for verification and validation of aspects. However in these studies the verification is not done with using base code or some mock classes, instead verification with respect to requirements and design of aspect are discussed. In these approaches the point of view is different from ours. They only work on enabling verification according to specifications, although we do not have any previously stated specifications.

Different types of analysis of aspect oriented programs are done in several studies such as [2] and [23]. Although they do not directly aim to enable verification of aspect oriented programs, they are analyzing aspects in detail.

In [24], a methodology for formally verifying aspect oriented programs is described. After the base classes and aspects are specified in a formal way separately, their models are integrated. Then the implementation level byte code is reverse engineered and the model of the output of the reverse engineering process is compared with the initially specified models. Instead of concentrating on the behavior of aspects as we did, in this study they verify the aspect oriented programs according to their general requirement specification.

A contract-oriented approach for verifying aspect behavior is analyzed in [25]. Pre-conditions, post-conditions and invariants are used in this approach. The aim is to show the correctness of the behavior after integrating aspects into base code. In addition to showing correctness they consider reasoning the behavior and structure of the aspect oriented programs. As most of other approaches, the aim is similar; however the approach is different from ours. They perform verification of the aspect

behavior affect on the aspect oriented programs by including base program in the analysis.

The approach in [26] is close to the one described in [24] considering their ideas. In [26] they first model the base code and aspect in Unified Modeling Language and generate a system. By performing a generation process, an algebraic Calculus of Communicating Systems description is obtained from the system. After that, state machines are generated from that system description. The behavior of the model is verified and tested by using the obtained state machines. There are several conversion processes to enable verification and testing in this approach and that is an obvious disadvantage. On the other hand, it is certain that if the conversion process is successful, the verification and testing processes would become very reliable by using state machines.

The approach in [27] is introducing a specification technique that enables writing modular specification for reasoning about the control impacts of aspects on base code. They implemented their approach for Ptolemy programs [28]. Although they state that it is possible to use approach for different programs if there are different implementations, the disadvantage of the study is its usability scope.

When we consider the approaches that are mentioned in this section of the chapter, it is seen that there are different proposals to apply verification on aspect-oriented programs. All of them have advantages and disadvantages when compared with each other. It is going to be seen in the next section that obviously our solution occupies a place among the approaches explained in this section. There are similarities and differences between our solution and the approaches that are mentioned here.

CHAPTER III

PROBLEM DEFINITION

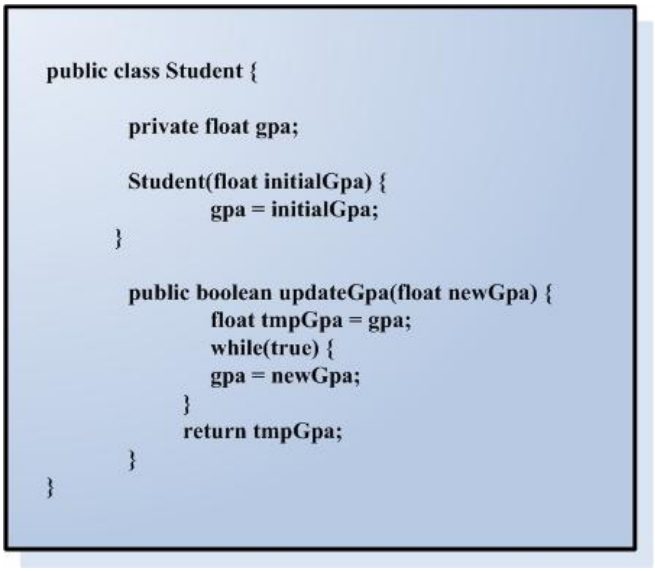
Although almost more than a decade has passed since the appearance of aspect oriented programs in software community and despite the fact that it is proposing a new modularization method to provide high cohesion, usage of aspect oriented programs fail short of meeting the expectations. One of the important reasons is the lack of established technology for reasoning about the reliability of aspect oriented programs. The two basic methods that stand in the forefront are testing and automated verification. Filling the gap for a verification framework of aspect oriented programming is an active ongoing research effort [14],[18],[19].

A common technique in program verification and testing is isolation of units and performing assume-guarantee reasoning. For this purpose, in this thesis, we focus on the question of how to isolate an aspect and encapsulate the aspect with its most general environment automatically. An automation tool is a necessity for achieving our goal and in the section below the details of that necessity is explained.

III.1 Need for This Study

In the first and the second chapters, it is stated several times that verification of an AspectJ programs is different from verification of other programs. The difference comes from the existence of two separate parts of an AspectJ program, because verification types vary when the number of parts of a program increases. In AspectJ, it is difficult to determine the reason or detail of a problem that is exposed during verification. In other words, growing functional interleaving among separate parts of software makes the verification process more complex. Another point of view is related with the reusability of aspects. It is more practical to check the aspect once separately and then use it with any base code. In order to get more specific results from verification and avoid state space explosion up to some point during verification process, it is necessary to work on the base code and the aspect code separately.

Consider the verification of the aspects in isolation. A developer needs to perform extra effort to write new classes, instances of which are behaving like the actual objects of base code. In other words the developer needs to verify the aspect code with mock objects of the base class objects. An example for explaining this process is as follows.



```
public class Student {

        private float gpa;

        Student(float initialGpa) {
                gpa = initialGpa;
        }

        public boolean updateGpa(float newGpa) {
                float tmpGpa = gpa;
                while(true) {
                gpa = newGpa;
                }
                return tmpGpa;
        }
}
```

Figure 25: Student Class

Assume that the `Student` class in Figure 25 is given and the developer is expected to write an aspect that works on this class. The aspect is supposed to keep the information about the activeness of a student and use it somehow in another method. We are not dealing with the usage of activeness information in this example. A hint about the activeness of the student is also supported: "If the `updateGpa` method of `Student` class is called at any time, student is considered as active".

In such a scenario as declared above, one way to present a solution is keeping an intertype variable in the aspect. An aspect as `StudentAspect` that represents this solution is shown in Figure 26.

```
public aspect StudentAspect {
        boolean Student.isActive = false;
        after(Student student)
        : call(public void Student.updateGpa(float))
        && args(gpa) && target(student){
                if(student.isActive)
                        proceed();
                student.isActive = true;
        }

        …
        // assume 'isActive' is used at here
        …
}
```

Figure 26: Student Aspect

An intertype variable namely `isActive` is used as a flag in the aspect and it is set to `true` when the `updateGpa` method is called and if `isActive` is not already set to `true` formerly. Then `isActive` variable is used later in the aspect code block.

After the base code and the aspect are ready, in order to verify the aspect code the developer has to run the verification of the woven code. Then, although the developer is responsible only for the correctness of the aspect code, compulsively it becomes necessary to check the correctness of base code as well. In order not to deal with the verification of base code, developer needs to write an extra class for only verification of aspect. A possible mock `Student` class may be the one provided in Figure 27. (This example is chosen to explain the topic as simply as possible; tricky examples, which are difficult to write mock classes manually, are provided in Chapter VI)

```
public class Student {

        public boolean updateGpa(float newGpa) {
                return randomBoolean();
        }
}
```

Figure 27: Simple Student Class

By writing the mock class in Figure 27, developer tries to eliminate the problems of base code during verification, and actually he is successful at this point. When we look at carefully to the below code block which is taken from the original base code of `Student` class, it is obvious that using mock class is necessary for pure verification of aspects.

```
while(true) {
                    gpa = newGpa;
}
```

Figure 28: Infinite While Loop

Since the shown while loop in Figure 28 is an infinite loop, developer would have to deal with this problem during verification, if he did not use the mock class of Student class.

After writing the aspect code and the mock class of base class, the last step for a developer to prepare the aspect for verification is writing a driver class (a class with `main` method) to call the base class methods for activation of point-cuts of the aspect. A possible driver class for the current example is shown in Figure 29.

```
public class StudentAspectEnvironment {
        public static void main(String args[]) {
                Student instanceStudent =  new Student();
                public float varGpa = randomFloat();
                instanceStudent.updateGpa(varGpa);
        }
}
```

Figure 29: Student Aspect Driver

The driver class creates an instance of `Student` class, and then initializes a random variable of type float to use as parameter of method `updateGpa`. Finally calls the method and then triggers the only advice in `StudentAspect` aspect. Thus, the

preparation process through verification is completed. After this process, usually by using a verification tool, the aspect with its environment is tested several times and the tool returns the results of verification.

In conclusion considering the process after the aspect was written, it is surely found that to only focus on the problems of aspect, some extra effort is necessary for a developer to perform. Basically, preparation of a mock class and an environment class constitutes the content of that extra effort.

III.2 Difficulties of Base Code Generation

Lightening the unnecessary load of developer during aspect verification by an automation tool is a good idea; however it is not easy to make that idea work. In this part of the chapter we focus on the difficulties of creation of such an automation tool, especially the difficulties of base code generation processes of that tool are listed.

- **Difficulty of Any Code Generation**

  Code generation can be a difficult subject on it's own to work on when we consider the software development subject areas. Actually, difficulty is more obvious if the generated code should conform to the rules of a traditional programming language [29]. Every detail of the language should be taken into consideration carefully. One of the most important points of correct code generation processes is handling the common syntactic issues of language. Syntaxes that are specific to AspectJ like dots, commas, and capital letters are expected to be handled painstakingly.

- **Difficulty of Satisfying Code Generation**

  This is the most critical issue among the difficulty issues that are explained in this part. In order to be sure that verification of aspect is done completely or without any problem, it is important to generate all necessary code for base part.

For example, all of the join points in the aspect should be enabled once or more than once during verification. While generating base code, these kinds of criteria are critical, although any missing of such points is not realized during verification process. As another example, since it is possible for values of base variables to change, it is not enough to assign fixed values to base variables. To achieve coverage, each variable should have every value that will affect the behavior of the unit. Because of such significant issues discussed in here, this is the most critical point that should be considered during base code verification process.

- **Difficulty of Efficient Code Generation**

Another difficulty point that is important for code generation process is efficiency of generated code. Efficiency term in here comprises the meanings of the concept of minimum cost from the view of line of code, efficient memory usage, and speed.

Generated code is not supposed to include unnecessary or dead code and every lines of code are expected to be generated for one purpose or more than one purpose while none of the two different code blocks are expected to be doing the same job during verification process of aspects.

For verification, the state space is a problematic issue. During verification all possible unique execution paths are exercised to realize a complete verification. If the code is written without any care about memory usage, then during the verification, state space explosion becomes inevitable. At base code generation process, in order to prevent a possible state space explosion, the code should be generated with memory usage consideration. From the memory usage view, the most important point is at initialization of objects. If it is possible, an object should be initialized once and then used many times, instead of initializing a new object every time.

The significance of this item is not as high as the former item's, however this is a more difficult item to handle.

CHAPTER IV

SYSTEM ARCHITECTURE

This chapter presents the architecture of the aspect environment generation system proposed in this thesis. Environment of an aspect is composed of the base code that it is weaved and the driver code that triggers it. Driver code is necessary for verification goal. On the other hand, existence of base code in generated environment is urgent, considering aspect oriented programming rules. Recall that an aspect oriented program is composed of two basic parts. An aspect and base code classes are those composing parts of an aspect oriented program. Interaction points of these two separate parts are called as join points (detailed information about them is provided in the Chapter II). Figure 30 visualizes the interaction of join points between the aspect and the base code.

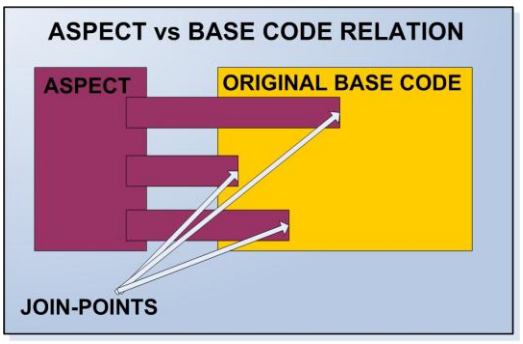


Figure 30: Aspect and Base Code Relation

Figure 31 visualizes the general work principle of the system with its most general architecture. There are two modes in the system. These modes are; environment generation for simulation and environment generation for verification. Generated code in verification mode enables the evaluation of all possible execution paths for verification tool. On the other hand in simulation mode as much possible values as possible are provided by the generated code.

In order to start code generation, two files are taken from user as input. One of them is aspect code and the other is the file that contains the data domain definitions of the variables that aspect uses while interacting with its environment. This second file is used only for the code generation for verification mode.

User interface redirects the input(s) to the 'Encloser' module of the core of project. 'Encloser' redirects the aspect file to JTB parser for parsing and keeps the variable domains file in hand. Then by traversing the syntax tree returned by JTB parser, 'Encloser' module fills the data objects that are specified in section V.2.3 in order to prepare the inputs of 'Generator' module. Then, 'Encloser' sends those objects to 'Generator' module by adding the variable domains file content as another parameter. 'Generator' module generates three outputs by using the inputs coming from 'Encloser' module. These three outputs are; base classes to interact with the aspect code, driver class to activate the join-points and domain-utility classes to enable verification and simulation goals. 'Generator' module sends these three outputs to user interface. Lastly, user interface displays these outputs to the user and saves them to the file system according to the user's command.

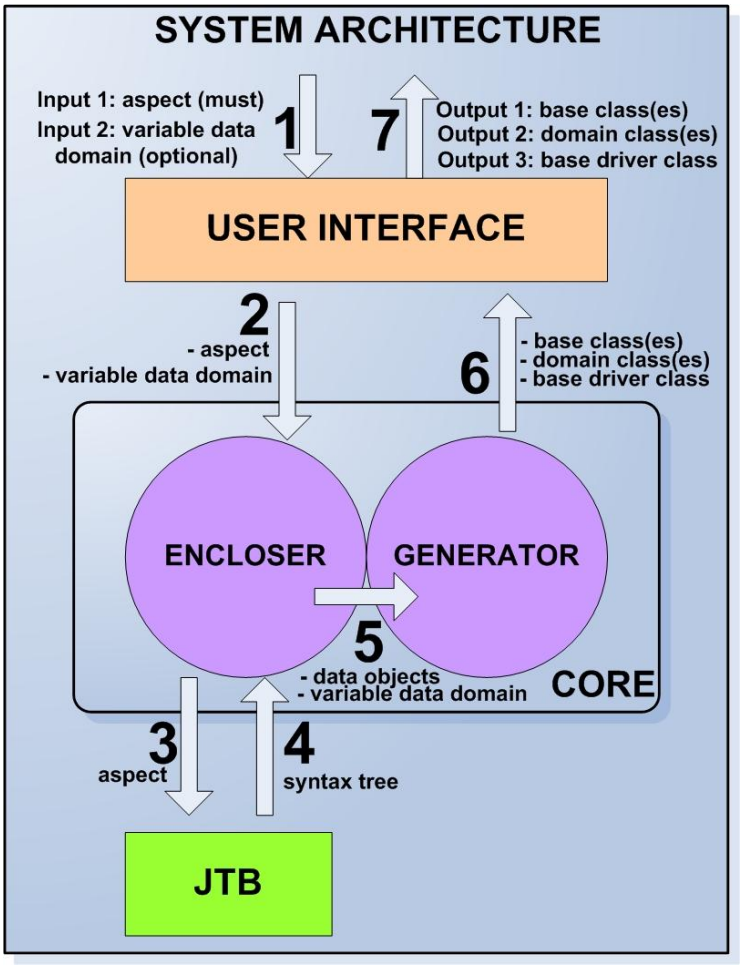


Figure 31: System Architecture

If we look at the key idea of our solution from the most general view, we aim to find the points that the aspect cuts the base code analyze it and then create a general environment for the aspect. Creating a general environment means to create base code that is providing all possible aspect-behavior affecting input values and that is triggering all possible aspect-behavior affecting points. Abstracting the aspect-unrelated parts of base is a side effect of this solution. Figure 32 displays an automatically generated base code structure and its consistency with the aspect. Base code is generated for the aspect visualization in Figure 30. The generated base code fits with the join points of the aspect, although there are some differences with the original base code.

While generating the base code, a utility class, a variable domains class and a driver class are also generated. Driver class is generated for enabling all of the join points for verification of the aspect. While the driver class is enabling the join points, all of the data values for variables that are specified in the variable domains file are enabled. By doing this, we consider that the code will be used as input of a verification tool. Variable domains file is not only used by the driver class but also by the generated base code. Utility class is a fixed class that is used for generating random values for different types of variables. Methods of utility class are called from base classes and driver class.

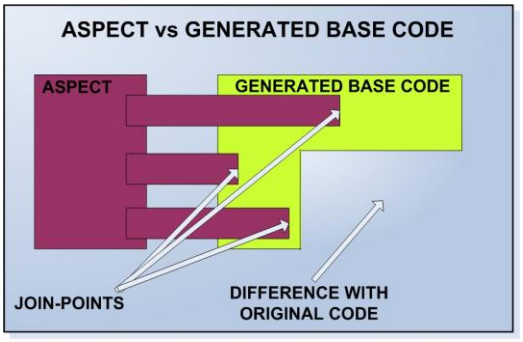


Figure 32: Aspect and Generated Base Code Relation

CHAPTER V

ASPECT ENVIRONMENT GENERATION

Through this chapter, the method that we propose for enabling the verification of aspects is discussed in detail. General system architecture of our solution is already represented in Chapter IV and in this chapter we are explaining them in detail with their logic. First, the rules that are determined for constructing the solution are listed with their explanations. Second, the data collection process that is performed by the 'Encloser' module is presented. Finally, code generation process of 'Generator' module is explained.

V.1 Rules of the Solution

Before starting to explain the technical view and details of the study, it is important to pay attention on the rules of the study. This section can be considered as the key section of the study, because it conducts the plan of technical part.

V.1.1 Make Filtering on Data Collection Process

Efficiency is very important in our study as any other software implementation. Considering the general structure in Figure 31, we should first clarify the step to filter the unnecessary code segments from the input aspect code. The necessary elements are discussed in the next rule; however this rule determines when to select necessary code elements. It is reasonable to make the filtering as early as possible, because unnecessary elements will be using the same extra effort required for the necessary elements during the process. Since parsing is handled by JTB parser, it is not possible to make filtering on the parsing sub-process of data collection process. It is suitable to make the filtering on data collection process of 'Encloser'. By this way the effort for unnecessary elements during the data collection and code generation processes are saved. Only necessary data objects are created during first step and only those objects are used during code generation.

V.1.2 Consider Verification and Simulation Purposes

One of the core work principles of our study is generating the code considering that it is going to be used for verification and simulation purposes. In light of the foregoing, the point-cuts of advices should be enabled with all possible inputs for verification mode. The principle is same for simulation mode except for 'all possible inputs' parts; it is 'as much as different inputs' for this mode.

V.1.3 Determine the Necessary Elements

This rule composes of a number of sub rules. The necessary elements for data collection and code generation are determined by this rule. While determining the necessary elements and their code generation processes, Rule 2 is taken into consideration. Rule 2 says that point-cuts should be activated with all possible inputs for verification mode and much possible inputs for simulation mode. To be able to

satisfy rule 2; each variable, which is read in any part of an advice should be assigned with its all domain values, because variable-reading points are out-effect points and they affect aspect behavior. These are the points where the advice gets input from its environment; therefore, these are the points where the environment can influence the aspect behavior. We handle these points by assigning all possible values; i.e., we model the affect of the environment by enabling all possible (at least behavior effecting) input values. On the other hand, a variable that is written in any part of an advice is not important for our case because these write operations are either internal to the aspect or they are an effect of the advice on the base code. In other words, these write operations are not input coming from the environment; hence, they do not have to be modeled and can be left as is. To make it more clearly, elements of an aspect are discussed one by one.

Below the sub rules of Rule 3 on necessary elements and their explanations are listed.

- Intertype Variables

  If an intertype variable is read in an advice code than that advice becomes dependent on that variable. Consider the example in Figure 33; it is possible for `Student` class to change value of `isActive` variable at any time. Therefore, to reason about the correctness of this advice, the advice should be enabled with all possible values of that intertype variable. Instead of dealing with this tiring process, the intertype variables are declared in base code as any other base variable and the procedure that is applied to base variables are applied to these intertype variables.

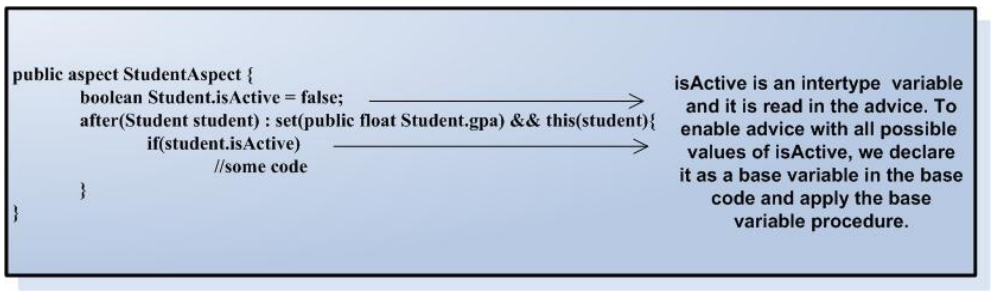


Figure 33: Student Aspect with Read Intertype Variable

If an intertype variable is assigned a value in an advice, in other words if it is written, that is not important for our case, because we do not model any case that is not an input coming from environment.

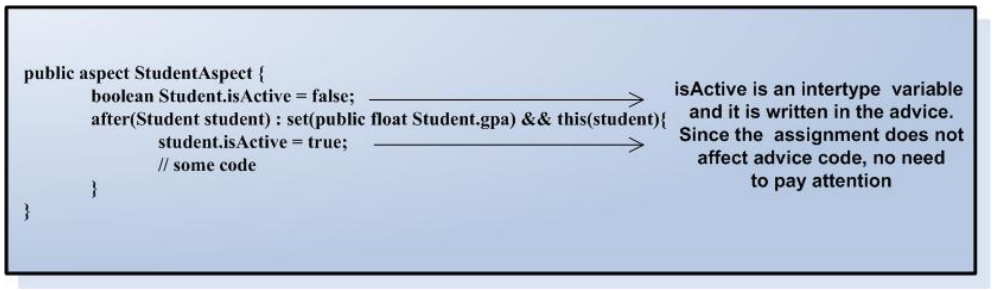


Figure 34: Student Aspect with Write Intertype Variable

- Base Variable

  Base variables, in the scope of this thesis, are the variables that are declared in the base code. Base variables are used as arguments of set-get type point-cuts and therefore they affect the behavior of aspect. Additionally base variables can be visible to aspects by the arguments of `args` and `target` keywords. While arguments of `args` correspond to arguments of method of point-cut, argument of `target` represents the actual object of the point-cut. As it is same for intertype variables, the reads of base variables need to be handled during code generation process and there is no need to deal with write base variables or assignments to base variables in advice code. If a base variable is read in an advice and that advice has a point-cut of type method call, then that base variable should be assigned with its domain values before activating the point-cut of advice. The following example code block demonstrates the situation more clearly.

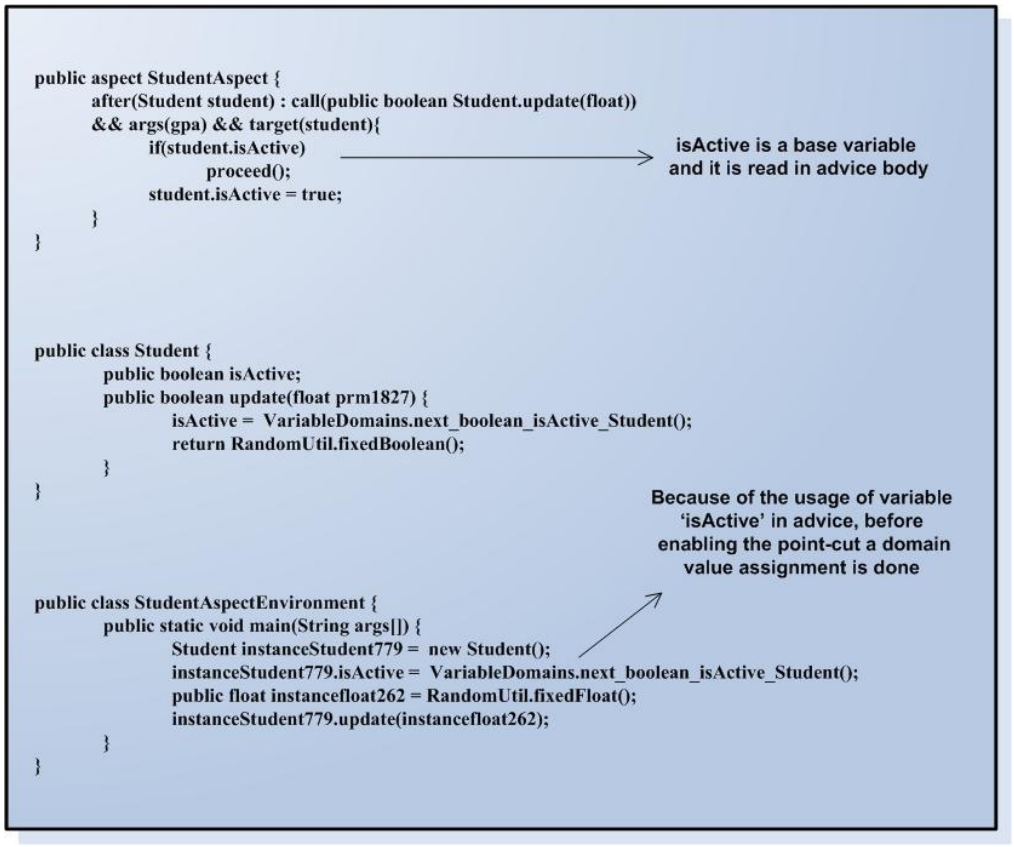


Figure 35: Student Aspect and Generated Code

The statements of `RandomUtil.fixedFloat();` and `VariableDomains.next_boolean_isActive_Student();` are going to be explained in the Base Code Generation section of this chapter.

- Base Method

  Base methods are referred in this document as methods that are declared and implemented in base code. Base methods are used in method call and execute type point-cuts of advices. Differently from intertype and base variables, all kinds of base methods are important for our study. Update method of Student class in Figure 35 is an example usage for base methods. Considering this example, since `update` method changes the value of `isActive` base variable and since base variables affect aspect behavior, we can conclude that base methods affect aspect behavior.

- Advice Body Elements

  Other than the base and intertype elements of aspects, there are also code blocks in advice bodies to consider during code generation. Therefore advice bodies are analyzed to extract the related elements and generate their base code. Some example analyses are shown in Figure 36.

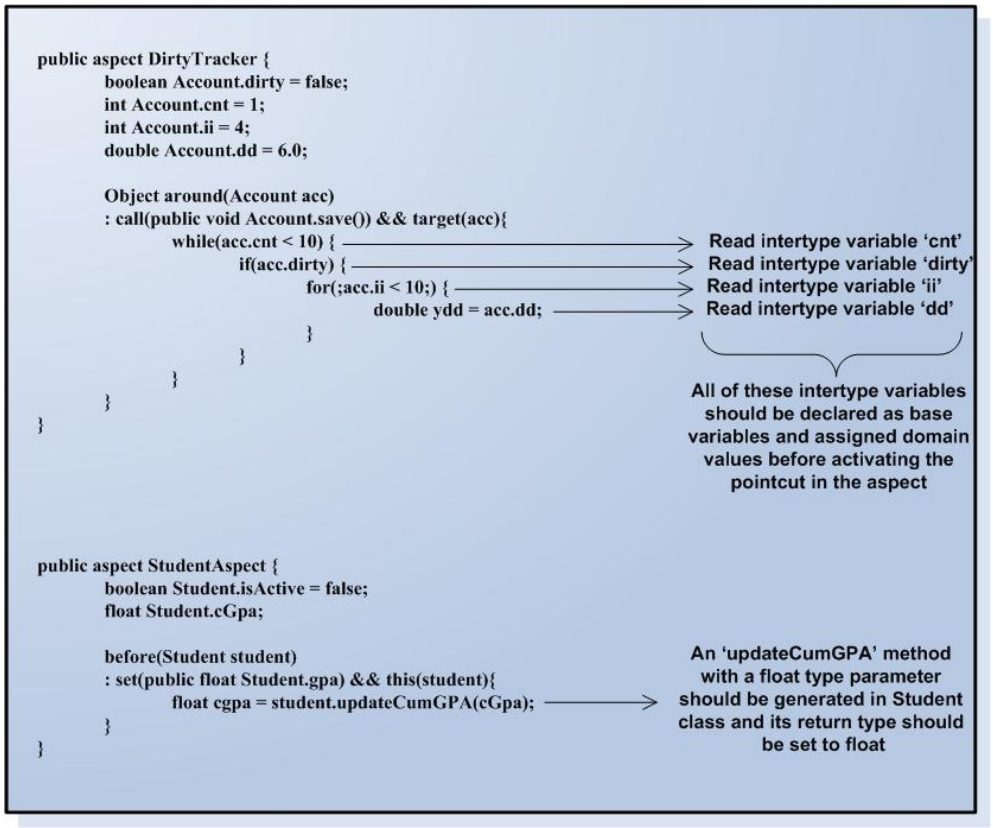


Figure 36: Advice Body Analysis

- Cflow Statements

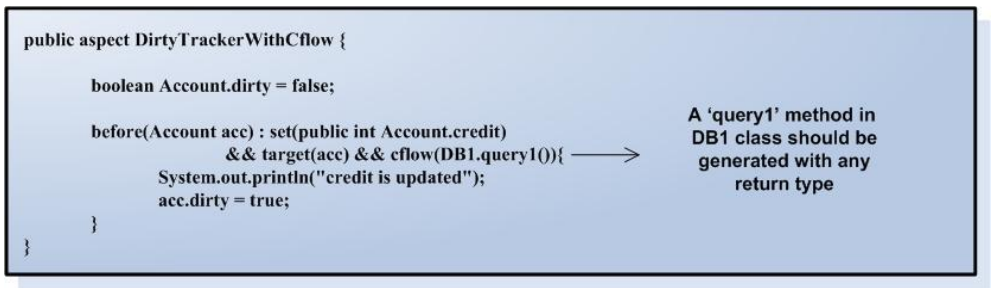


Figure 37: Cflow Usage

`Cflow` keyword in AspectJ is the abbreviation of control flow. It is easier to explain it on an example. Considering the example `DirtyTrackerWithCflow` aspect in Figure 37, a statement stands out as `cflow (DB1.query1())`. This statement means that if an assignment to the `credit` variable of `Account` class is done in the control flow of the `query1` method of `DB1` class, then the point-cut of the advice should be activated, otherwise the assignment event should be ignored. To put the meaning and usage of `cflow` aside, here the focus is on the need for a `query1` method of `DB1` class. As any other elements to be generated in the base code, the code for the declaration of `query1` method of `DB1` class should also be generated.

In addition to the methods, it is possible for `cflow` statements to take point-cuts as inputs [12]. This fact makes the handling of the `cflow` statements very complicated. Actually, considering the aim of our study it is not a critical issue to enable the point-cuts in the control-flow a method. It is critical to enable point-cuts independent of the point that it is enabled. Therefore, our tool does not support `cflow` statements; however, `!cflow` usage is supported. It is not necessary to worry about control-flow condition for `!cflow` usage, because we are sure that it is not possible for the generated method (that we enable the point-cut in) to be the parameter of `!cflow` statement.

V.2 Data Collection Process

To be able to generate a meaningful and functional Java code for verification, firstly the aspect code needs to be parsed successfully. Since parsing is a tedious task and there are some open source applications that are already in use, we prefer to use a suitable parser for parsing the aspect code. Although there are different parsers to be used for AspectJ, the best one to integrate to the project is the one that is generated by using JTB. It is specific to Java programming language, it is used widely and therefore there are several resources about it, it is easier to understand and to use

compared to other tools [7]. Considering these reasons, we have chosen JTB parser for parsing and have modified the sample grammar file in it to make it possible to handle AspectJ code. By parsing AspectJ code we collect data to fill the data objects that are explained in section V.2.3.

To have a reusable design and application the 'Data Collection Process' part is constructed by applying the strategy design pattern [30]. Before explaining the data collection process of our solution in detail, first the design of this process is explained.

V.2.1 Design of Data Collection Process

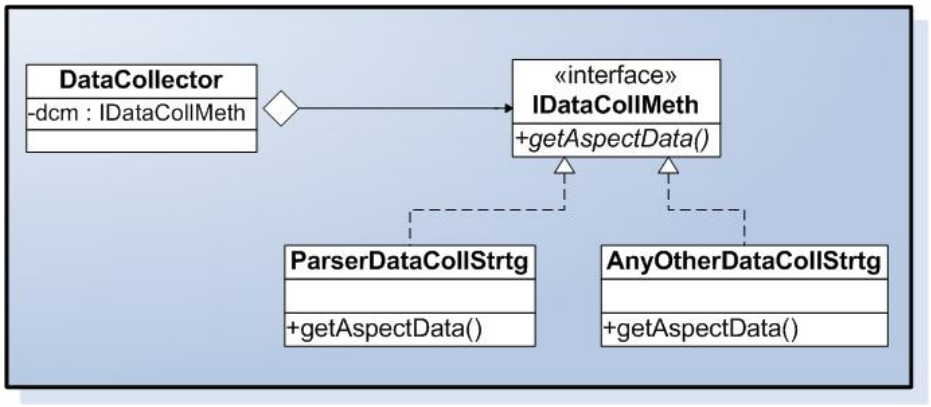


Figure 38: Data Collection Process Design

Since we want a solution that is open for extension, for the data collection process of our solution, we have chosen to use the strategy design pattern [30]. Although, at the scope of this study there is only one data collection methodology, for the future work and possible changes of the study, we have designed our solution by using this pattern. Considering Figure 38, `DataCollector` is our application and it corresponds to `Context` in strategy design pattern. `IDataCollMeth` is the interface of strategies that are used for data collection process. `ParserDataCollStrtg` is the concrete strategy that is in use. If a new methodology is needed, the strategy class of that methodology is prepared and set to the main application as shown in Figure 38.

V.2.2 Parsing

We analyze the parsing process of the AspectJ code under 2 sub-topics as ‘Additions to JTB Grammar’, which explains the additions to the grammar in this study, and ‘JTB Syntax Tree’, which focuses on the syntax tree output of the JTB and its analysis.

V.2.2.1 Additions to JTB Grammar

In the standard distribution of JTB, a Java grammar is provided as a sample grammar [7]. To use JTB parser for parsing AspectJ code, the sample grammar of JTB needs to be extended to recognize AspectJ specific statements. The necessary syntax rules for AspectJ language is obtained from [31]. Based on these rules, the grammar is modified and the parser is recompiled with the new grammar file generating new syntax tree nodes and visitors that will traverse the syntax tree (see Section II.3). Normally, the generated JTB parser is expected to be faultless however there were some faults in the generated parser and those faults are corrected manually. The newly added syntax rules, which are elements of AspectJ programming language, are given in Appendix B. The first one, `AspectDeclaration`, is explained in detail here as an example.

```
AspectDeclaration
{
ModifiersOpt() ["privileged" ModifiersOpt()] "aspect" <IDENTIFIER> SuperOpt() InterfacesOpt() PerClauseOpt() AspectBody()
}
```

Figure 39: Aspect Declaration

`AspectDeclaration` text block in Figure 39 is added to make the parser aware of this syntax rule between the curly braces. An example about the usage of syntax rules is provided in the following section. Detailed information about JTB and the grammar used by it is provided in Chapter II.

In this example most of the elements that are constructing the `AspectDeclaration` syntax are already defined in sample JTB grammar. One of them, `ModifiersOpt()`, is a known syntax in JTB grammar and we have used this element as a part of our `AspectDeclaration` element. In the definition of `AspectDeclaration` only `AspectBody` is an unknown element for the existing grammar. We have defined this element as shown in Appendix B and we have used it at here as a part of `AspectDecleration`.

Just like `AspectDeclaration` the other elements in Appendix B are defined in the grammar file and the parser is generated according to that grammar.

V.2.2.2 JTB Syntax Tree

JTB takes a grammar file as input and produces a syntax tree, a depth first visitor class to traverse that syntax tree, and JavaCC grammar with embedded Java code. For customizing the tool and data collection from the parsing process, the users extend the default visitor class generated. By extending the visitor class we overwrite the parts that define the rules which we are of our concern and collect only the data that we need. To understand it more clearly, consider the following example.

```
public aspect StudentAspect {
        boolean Student.isActive = false;
        //Advice
        /* @ ensures
        @ student.isActive == true;
        @*/
        after(Student student) : set(public float Student.gpa) && this(student){
                System.out.println("gpa is updated");
                student.isActive = true;
        }
}
```

Figure 40: Student Aspect

Assume that we have an AspectJ file as student.aj with the content as shown in Figure 40. `StudentAspect` is an aspect with an intertype variable called `isActive` and an advice. The advice is a `before` type advice. The point-cut expression in the

advice is dealing with the `gpa` variable in the `Student` base class. When the value of `float` type `gpa` variable is changed, the advice is activated. To explain the syntax tree only the advice is considered in this example. The root of the syntax tree for this code segment is an `AspectDecleration` node. Recall that the aspect declaration rule is as in Figure 39. Figure 41 gives the derivation of this rule for the given example.

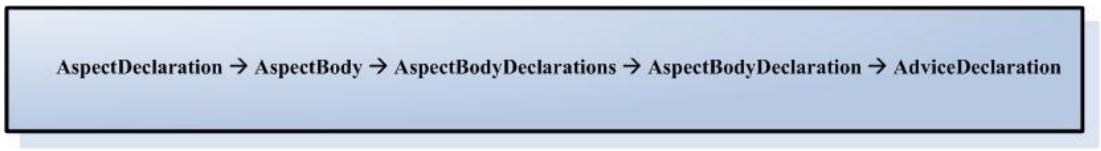


Figure 41: Aspect Declaration

In the grammar file we declared that `AspectDeclaration` includes an `AspectBody`, `AspectBody` includes an `AspectBodyDeclarations` and so on. After the "student.aj" file is given to JTB as input, it returns a syntax tree as output. By traversing the output syntax tree we can find `AdviceDecleration` and collect the data that we need to generate the aspect environment. Details of all these processes are explained by an example at Appendix C.

V.2.3 Collecting Data

After the basics of syntax tree traversing are determined, there is a need for the data types to load the collected data from the tree. In this section these data types and data collecting methodology are explained.

V.2.3.1 Data Types

An example class diagram is shown in Figure 42 to simply explain the structures of data types.

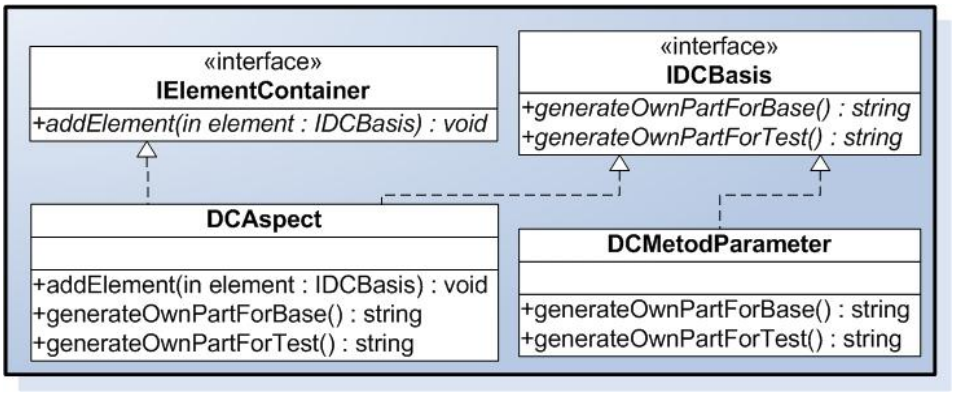


Figure 42: Example Class Diagrams of Data Types

As it is seen in the class diagram, all of the data types are implementing the `IDCBasis` interface. Details of the methods in `IDCBasis` are explained in section V.3 while explaining the base code generation sub-topic.

The data types and their brief explanations are listed below. `DC` affix in front of the class names represents that the class is a Data Container.

- DCAspect

  This is the highest level container. It may contain `DCAdvice`, `DCBaseVariable`, `DCIntertypeVariable` or `DCBaseMethod` type other containers.

- DCAdvice

  `DCAdvice` represents the data type of advices in aspects. It may contain `DCPointcutTypeSet` or `DCPointcutTypeCall`.

- DCBaseVariable

  This data type exists for base variables. It is one of the lowest level data types and it does not contain any one of containers, listed in here.

- DCIntertypeVariable

This data type exists for intertype variables. It is one of the lowest level data types and it does not contain any one of containers listed in here.

- DCBaseMethod

`DCBaseMethod` represents the data type of base methods. It may contain any number of `DCMethodParameter` type containers.

- DCMethodParameter

This data type exists for keeping data of method parameters. It is one of the lowest level data types and it does not contain any one of containers listed in here.

- DCPointCutTypeCall

`DCPointCutTypeCall` data container is for the point-cut types other than `set`, such as `call`, `initialize`, `executions` and so on. It may contain `DCPointCutParameterTypeCall`, `DCBaseRW`, `DCIntertypeRW` types of containers.

- DCPointCutParameterTypeCall

This data type represents parameters of point-cuts that are not type of `set`. It may contain any number of `DCMethodParameter` type data containers.

- DCPointCutTypeSet

`DCPointCutTypeSet` data container is for the point-cut types of `set`. It may contain `DCPointCutParameterTypeSet`, `DCBaseRW`, `DCIntertypeRW` types of containers.

- DCPointCutParameterTypeSet

This data type represents parameters of point-cuts that are type of `set`. It is one of the lowest level data types and it does not contain any one of containers listed in here.

The `RW` affix in the following two data types is abbreviation of read-write. These two data types are used for keeping the data of written or read, base or intertype variables in advice bodies.

- DCBaseRW

This data type exists for written or read base variables in advices. It is one of the lowest level data types and it does not contain any one of containers, listed in here.

- DCIntertypeRW

This data type exists for written or read intertype variables in advices. It is one of the lowest level data types and it does not contain any one of containers, listed in here.

V.2.3.2 Methodology of Collecting Data

JTB provides us a class for obtaining the syntax tree. `DepthFirstVisitor` is that class and to use syntax tree, we need to overwrite the necessary "visit" methods of this class in our class, which is extending `DepthFirstVisitor` [7]. The code block in Figure 43 explains more clearly the usage of syntax trees.

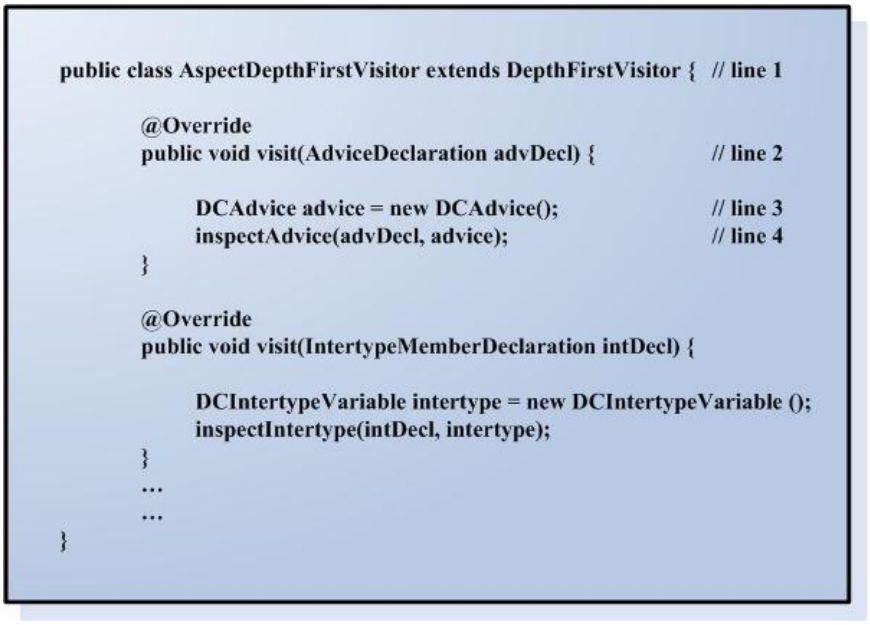


Figure 43: Extending Visitor Class

The class `AspectDepthFirstVisitor` is extending the `DepthFirstVisitor` class that is generated by JTB as shown in line 1. In this example, as it is seen from line 2 to line 4, the `visit` method of `AdviceDeclaration` is overwritten. A data object of type `DCAdvice` is created and sent to the method `inspectAdvice` as the second parameter. The method takes the caught object of type `AdviceDeclaration` as the first parameter and the `DCAdvice` type object as the second parameter. The method traverses the first argument and fills the second argument. Just like the `AdviceDeclaration` type, other necessary types are visited and the data objects are filled.

V.3 Base Code Generation

After the parsing and data collecting processes are completed the next step is code generation. Parsing and data collecting are means to go code generation, and our actual goal is generating the environment of aspect. In this section, details about default methodology of code generation process in our study and variable domain assignment points are discussed.

V.3.1 Design and Details of Code Generation

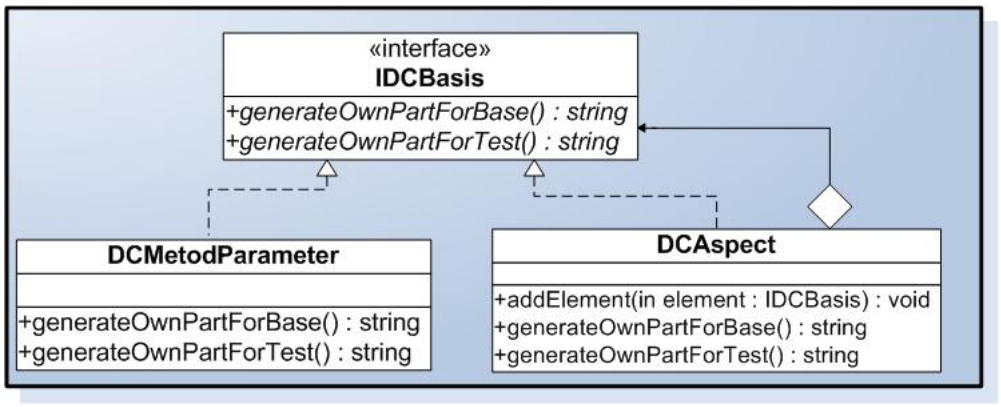


Figure 44: Code Generation Process Design

As it is described in section V.2.3, there is a containment relationship among the data objects that we use and this property makes composite design pattern usable for us [30]. `IDCBasis` interface of our design in Figure 44 corresponds to `IComponent` participant of the composite design pattern. Similarly `DCAspect` is corresponding element of `Composite` item while `DCMethodParameter` is the same for a `Leaf` node of composite design pattern. The idea in here is that, code generation process is embedded into the data objects and they generate only the codes that they are related with. There are two methods in `IDCBasis` interface. The first one `generateOwnPartForBase` method of a data object generates the environment code that is related with the object except for related codes in driver class. Then if the data object is a composite one like `DCAspect`, this method calls the same method of each data object that it contains. `generateOwnPartForTest` works similarly, the only difference is that this method generates only the driver class code piece of that data object.

To explain clearly, it is better to follow an example for code generation process. `DCAspect` is the out most enclosing data type and it may contain `DCAdvice` type objects. `DCAdvice` type objects may contain `DCPointCutTypeCall` type objects and `DCPointCutTypeCall` objects may contain objects of type `DCPointCutParameterTypeCall`. Finally, `DCPointCutParameterTypeCall`

objects may contain `DCMethodParameter`. The aspect in Figure 45 is prepared to explain this code generation process and the generation flow from `DCAspect` to `DCMethodParameter`.

```
public aspect StudentAspect {                                          // line 1
        boolean Student.isActive = false;                              // line 2
        after(Student student) : call(public boolean Student.update(float))
        && args(gpa) && target(student){                               // line 3
                if(student.isActive)                                   // line 4
                        proceed();                                     // line 5
                student.isActive = true;                               // line 6
        }                                                              // line 7
}                                                                      // line 8
```

Figure 45: Student Aspect

From `StudentAspect` aspect the code blocks represented in Figure 46 are generated with our tool. The `Student` class is the only class in base part of the generated code, and `StudentAspectEnvironment` class exists for the verification process of the aspect with the generated base code.

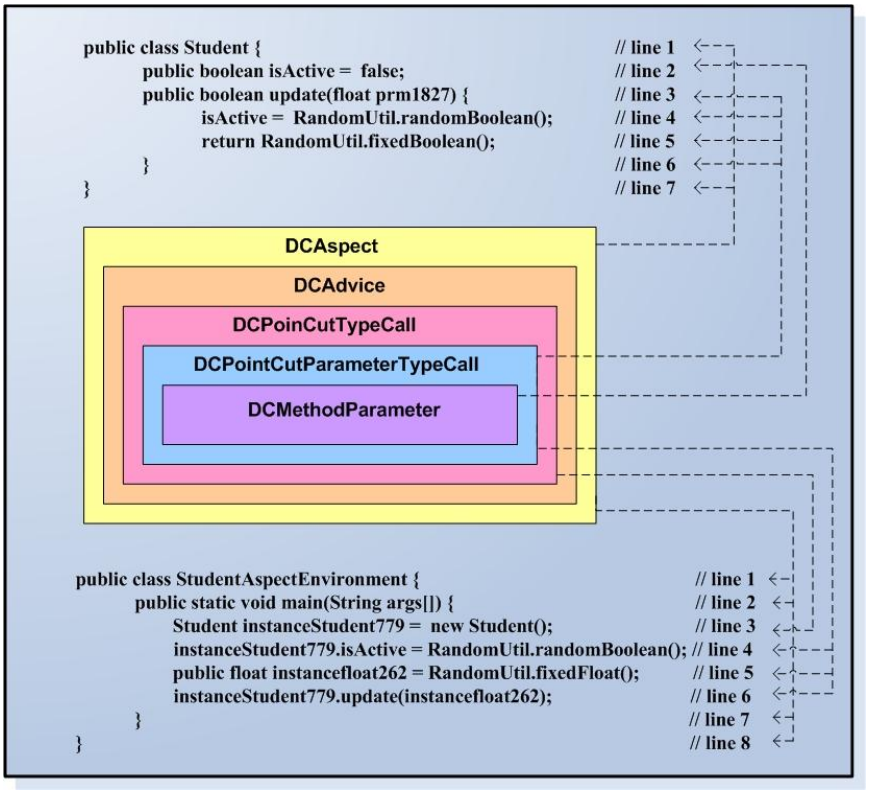


Figure 46: Working Style of Composite Design Pattern in the Solution

By the tool for every aspect at least one base class and exactly one driver class (for verification) is generated. For this example the only base class is `Student`. In this example we focus on the point-cut part of the advice. In other words line 1 and line 3 of `StudentAspect` aspect is important for us. Before starting, it is considered that the parsing process is completed already and the data objects (`DCAspect` etc.) are ready.

The data object `DCAspect` produces line 1 and line 7 of Student class, and then line 1, line 2, line 7, line 8 of `StudentAspectEnvironment` class. After it finished its generation process, `DCAspect` object calls the code generation methods of the data objects in it and adds that insider code between its lines of code. Although there is an intertype variable we do not deal with it in this example and we focus on the `DCAdvice` object in `DCAspect`. Since we go on our example with the point-cut part of the advice, the advice does not generate any code related with this example and directly calls the code generation method of the `DCPointCutTypeCall` object in it.

DCPointCutTypeCall object does not generate any code in the Student class and calls directly the method of DCPointCutParameterTypeCall. In the StudentAspectEnvironment line 3 is generated by DCPointCutTypeCall, however details of this line of code are discussed in the next section. Except for the method parameter in line 3, the lines 3, 4, 5, 6 of Student class are generated by DCPointCutParameterTypeCall. Similarly, except for the method parameter of StudentAspectEnvironment at line 6, the lines 4, 5, 6 of StudentAspectEnvironment are generated by the object of type DCPointCutParameterTypeCall. The method parameters in both Student and StudentAspectEnvironment are generated by the data object of type DCMethodParameter in DCPointCutParameterTypeCall.

V.3.2 Variable Domains

In this section the variable domains used in code generation process is discussed. The technical and algorithmic views are presented in addition to general view of the usage. The necessity of variable domain assignments and usages arise from more than one reasons. Firstly, these reasons and their related usages are discussed with some examples. Then, in the second part technical information about variable domain assignments and usages are provided.

V.3.2.1 Type of Variable Domains Usages

Recall that one of our goals is simulating the inputs of aspect that are coming from outside. It is impractical to assign all possible values; therefore we aim to assign all values that can affect aspect behavior. There are three points that we can model the outside effects by using these values. First one arises from base and intertype variable usages inside advice bodies whose point-cut part is type of ‘call’ or ‘execute’. The second usage comes out at ‘return’ statements of methods, inside the generated base classes, and the last one is about calling methods with random parameters. For a general example, an aspect and the generated code from it are

shown in Figure 47. Then the three types of usages are explained by pointing the example.

```
public aspect StudentAspect {
        boolean Student.isActive = false;
        after(Student student) : call(public float Student.update(float)) && args(gpa) && target(student){

                if(student.isActive)
                        proceed();

                student.isActive = true;
        }
}

public class Student {
         public boolean isActive =  false;
         public float update(float prm1827) {
                  isActive=VariableDomains.next_boolean_isActive_Student();      →        //point1
                  return RandomUtil.fixedFloat();                                →        // point 2
         }
}
public class StudentAspectEnvironment {
         public static void main(String args[]) {
                  Student instanceStudent779 = new Student();
                 instanceStudent779.isActive = VariableDomains.next_boolean_isActive_Student(); → //point 3
                  public float instancefloat262 = RandomUtil.fixedFloat();                       → // point 4
                  instanceStudent779.update(instancefloat262);                                   → //method call
         }
}
```

Figure 47: Student Aspect and Its Environment

- If the advice of a ‘call’ type point-cut uses base variables inside its body, it means that potentially the code of body is dependent of the base variable value. Since we consider the intertype variables as base variables to facilitate the code generation process, both intertype variables and base variables are considered in the scope of this item.

  Considering the example code in Figure 47, before calling the `update` method, which is marked with `method call` label, an assignment is done to `isActive` variable. This assignment is focused with `point 3` label. Details of right hand sight of the assignment are going to be discussed in following sections; however, here basically it is enough to see that a domain value is

assigned to an intertype variable before calling a method that is using it in its body.

Since it is possible for the method `update` to change the value of variable `isActive`, at line that is marked with label `point 1`, a domain value assignment to the intertype variable `isActive` is done. This is a similar assignment to the one explained in the former paragraphs.

- The second usage of domain values is focused with `point 2` label in the example. Since the return type of method is float, a fixed value is returned to produce faultless code. This fixed value actually models the behavior of the base code. If return value of method is not a primitive type, an instance of the class is created and returned as the return value of the method.

- The third or the last usage style is shown in the line that is marked as `point 4` in the example code. Since `update` method of `Student` class takes an argument of type float, a variable of that type is created and given as input to the method at the next line.

Looking from the verification view; although it seems like an assignment of a domain value just one time, it is not, because verification tools, such as JPF, make all tries on the same line of code until all domain values of related variable that is specified by the user are finished and thus instead of a single domain value usage, it becomes an exhaustive usage. Domain value usage styles are explained with examples at this part.

V.3.2.2 Domain Value Assignment Algorithm and Methodology

As in the data collection process of our study, for the variable domain usages, strategy pattern [30] is used. A variable domain strategy of the code generation process is determined according to the code generation mode choice of the user. If

the user chooses verification mode code generation then `VerificationMode` strategy is created and set to application; if simulation mode is chosen then `SimulationMode` strategy is created and set to application. Contents of utility classes and variable domain value assignments are changed according to these strategies. If there is any need for a different methodology, it is easily applicable.

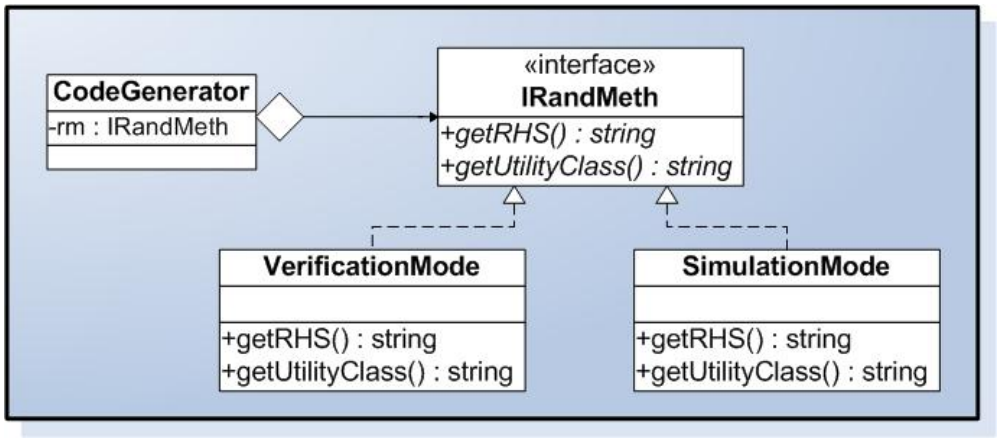
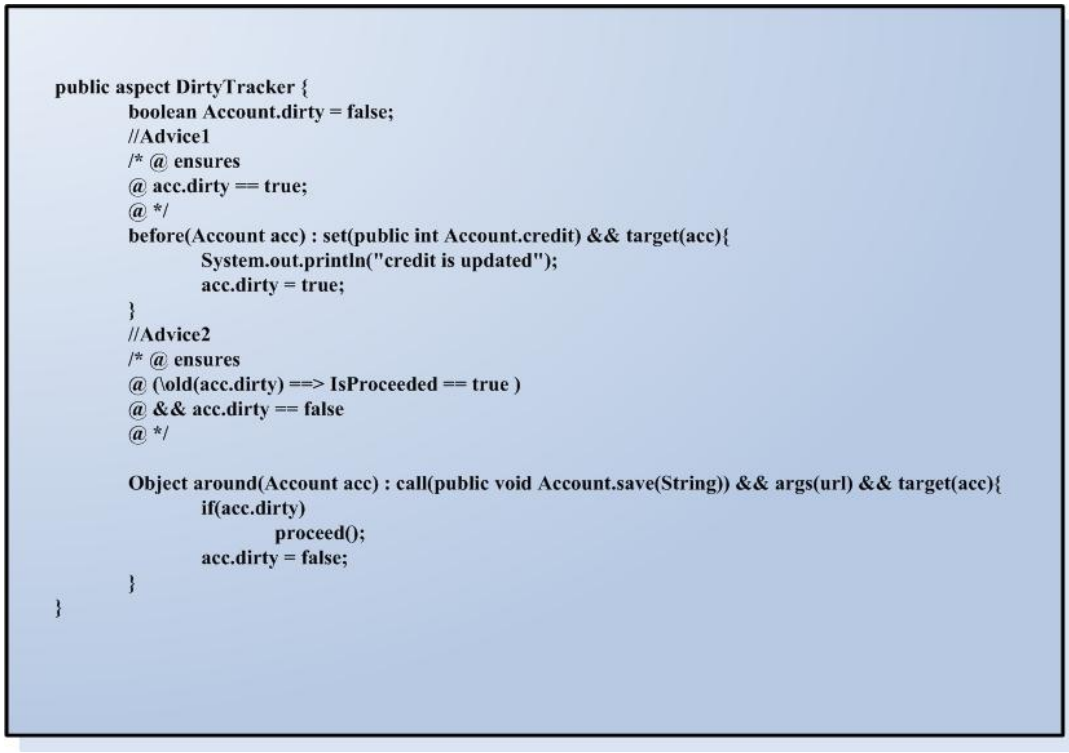


Figure 48: Application of the Strategy Design Pattern for Variable Domains

The strategy produces a right hand side value when the `getRHS` method of it is called and then this string is used by the context. `getUtilityClass` method of the strategy produces the utility class that is used for variable domains assignments in generated code.

For the aspect that is shown in Figure 49 both the code generated in Simulation mode and Verification mode are analyzed after their brief explanations.



```
public aspect DirtyTracker {
        boolean Account.dirty = false;
        //Advice1
        /* @ ensures
        @ acc.dirty == true;
        @ */
        before(Account acc) : set(public int Account.credit) && target(acc){
                System.out.println("credit is updated");
                acc.dirty = true;
        }
        //Advice2
        /* @ ensures
        @ (\old(acc.dirty) ==> IsProceeded == true )
        @ && acc.dirty == false
        @ */

        Object around(Account acc) : call(public void Account.save(String)) && args(url) && target(acc){
                if(acc.dirty)
                        proceed();
                acc.dirty = false;
        }
}
```

Figure 49: Example Aspect Code

- Simulation Mode Strategy

This strategy exists for simulation purpose. Instead of a user specified domain, in this strategy, a static random value utility class is used and the variables are assigned with the methods in the utility class according to its type. Since in this mode the goal is not full exploration of all possible execution, the program is going to be run just once for a single path. Therefore, state space explosion is not a problem and there is no need for the user to supply domain constraints. The static random utility class is given in Appendix A. Sample code blocks from the related utility class are shown in Figure 50.

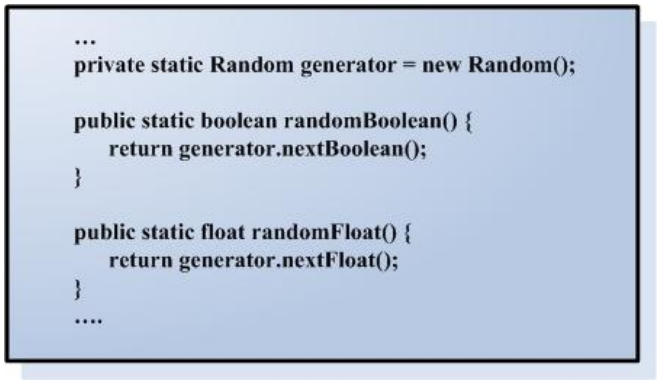


Figure 50: Random Utility Class

Considering the `DirtyTracker` aspect in Figure 49, the code that is displayed in Figure 51 is generated in Simulation mode.

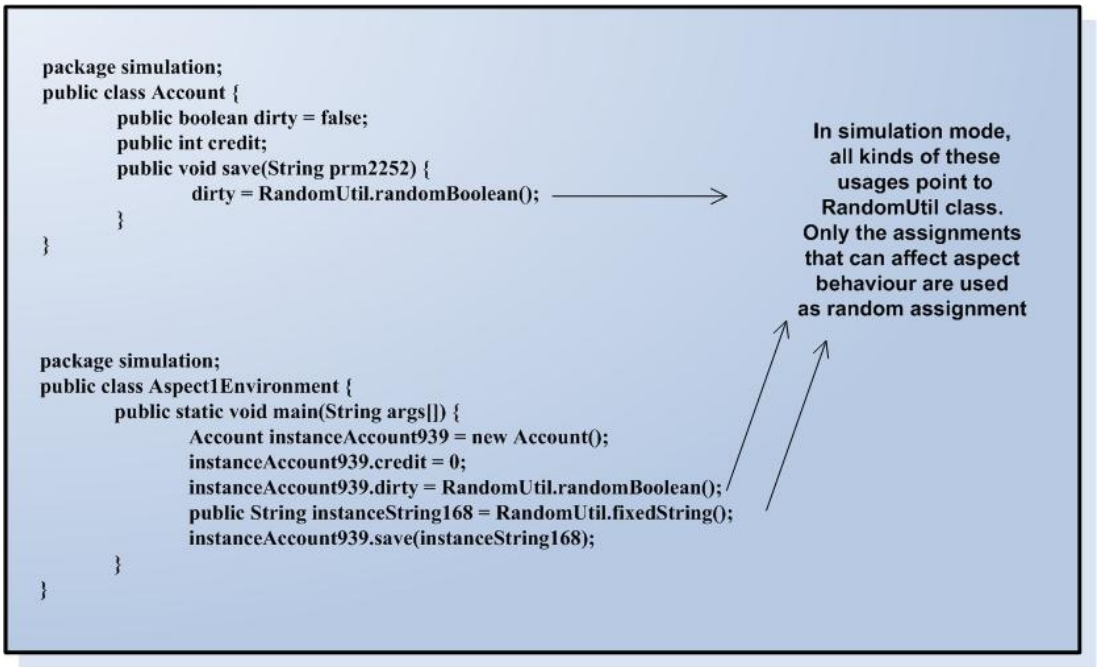


Figure 51: Generated Code Blocks for Simulation Mode

- Verification Mode Strategy

Verification mode is used for preparing the base code suitable for verification. Difference from the simulation mode is the effect of user for variable domains. Within a file, user prepares a list of base and intertype variables with their data domains and gives that file as input to our tool. Then when the code generator

needs to make an assignment to one of those variables, it uses a `VariableDomains` object that contains the information given in that file. The implementation of the `VariableDomains` class is prepared to make JPF verification tool possible to try all possible values of the variable in its domain. We ask the user to provide the domain values since all possible valuations, as in the simulation mode, makes the verification impossible due to practically unlimited number of combinations and backtrackings.

When we analyze the generated code in Figure 52 for the same aspect in verification mode, we see that the `RandomUtility` class is still in use at some points. `RandomUtility` class is used at the points that do not have an effect on aspect behavior and that it is not possible for the user to define domain values for; on the other hand that it is necessary to make an assignment to have syntactically correct code. The `instanceString317` variable of `Aspect1Environment` in Figure 52 is an example for this type of usage. `InstanceString317` is a variable that is automatically generated during code generation process and it is not possible and not necessary for the user to specify its domain values. Therefore `RandomUtility` class is necessary for Verification Mode Strategy.

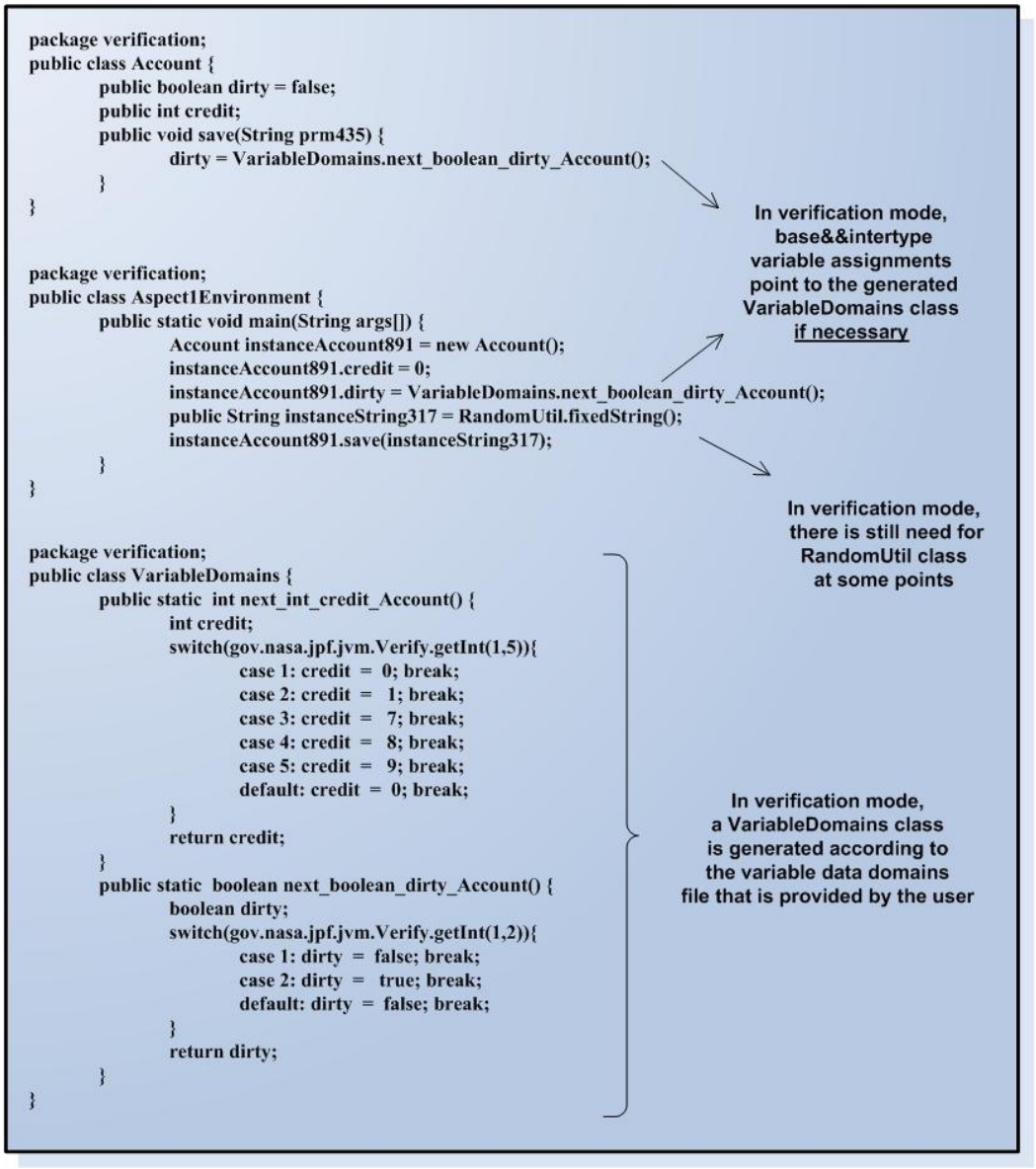


Figure 52: Generated Code Blocks for Verification Mode

Elements of the input file content that is provided by the user should be in the following format.

```
Type_VariableName_ ClassName
```

For example, a `myType` type variable `myVar` that is defined in `myClass` class should be added to the input file like `myType_myVar_ myClass = …`

Considering the example in Figure 52, we understand that for `credit` base variable the user placed the following statement in the variable domains file.

```
int_credit_Account = 0, 1, 7, 8, 9
```

V.4 Final Product: Aspect Base Code Generation Tool

Data Collection and Base Code Generation processes are the subjects that expose the structural side of the study in detail. After investigating the technical view of the study, in this part of the document a user level introduction about it is provided. User interfaces are used to analyze the tool step by step.

Figure 53: Main Window of the Tool

The window that is shown in Figure 53 is the main window of the tool. By using this main window, user may access other windows.

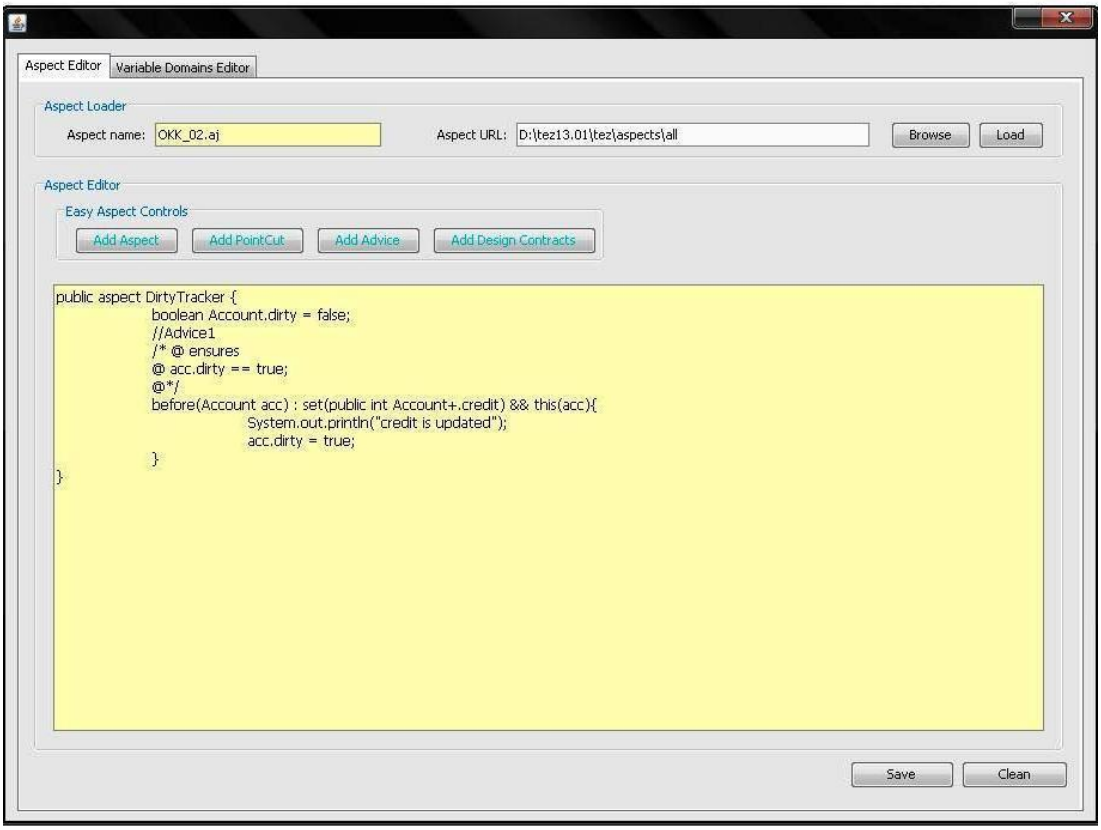


Figure 54: Editor - Aspect Editor Tab

By clicking the `EDITOR` button from the main window, user accesses the window that is shown in Figure 54. This window is an editor for users before starting to code generation process. By using the `Aspect Editor tab` that is illustrated in Figure 54 and by using the `Variable Domains Editor tab` that is illustrated in Figure 55, user can create new files or edit the existing ones.

First tab is used for creation and editing of aspect codes. By using `easy aspect control buttons`, user can easily make additions to the code. If the content of an existing file is needed to be investigated or it is needed to be updated then firstly it should be loaded from the `Aspect Loader panel` at the top of the tab. If a new aspect is going to be created than this panel is not used until saving process of the file. File saving process works independently from the purpose of editing or creating. When the user clicks `Save` button the tool saves the content of the `Aspect Editor`

panel into the Aspect URL directory that is provided in the Aspect Loader panel and the file is saved with name that is written in the Aspect name text field in the same panel.

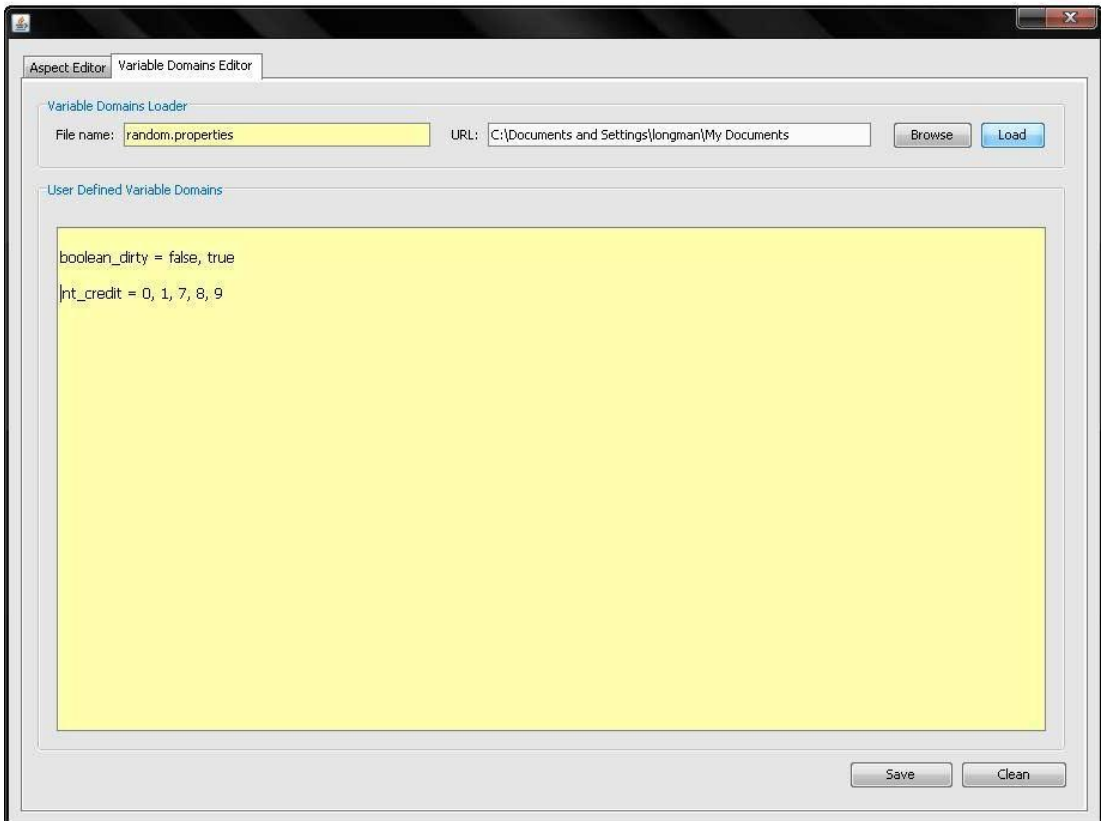


Figure 55: Editor – Variable Domains Editor Tab

Just like the Aspect Editor tab, Variable Domains Editor tab is used for creating, editing and saving. This time the argument is not an aspect, but it is a Properties file in which the domains of variables in the code are specified by the user.

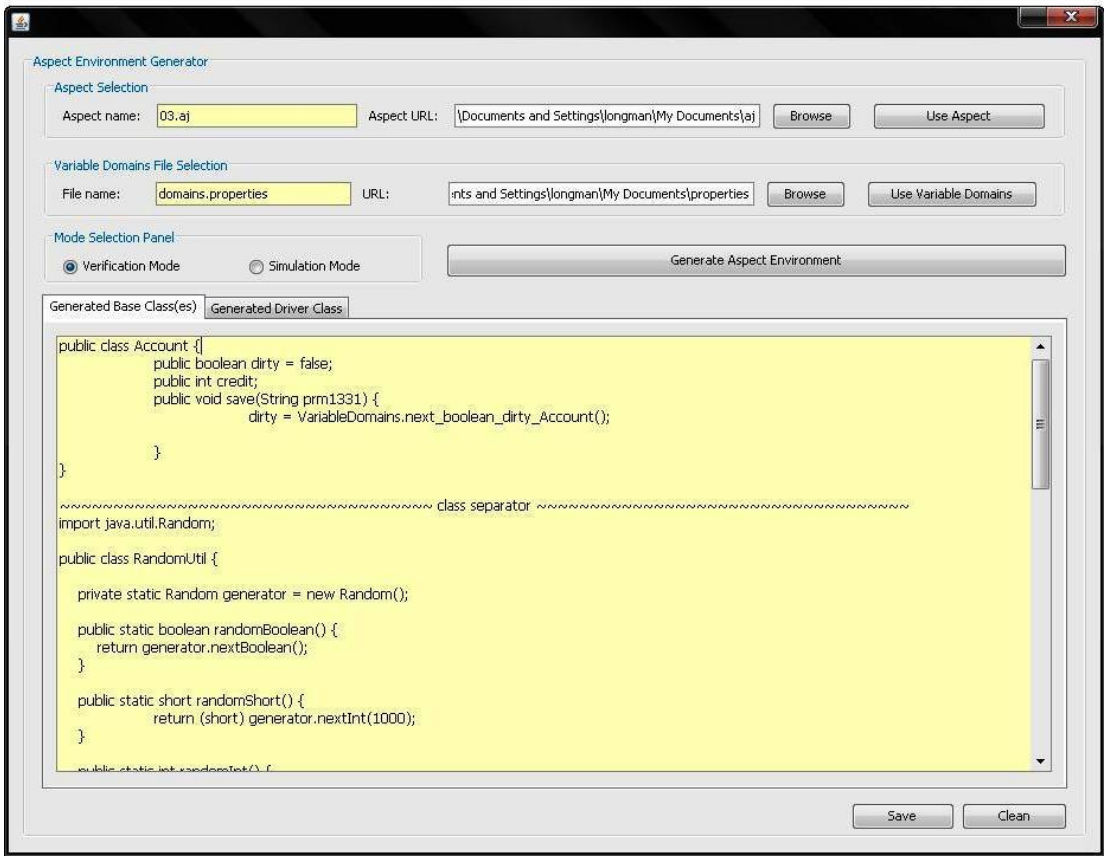


Figure 56: Base Code Generation Window – Base Classes –

From the base code generation window, the user should load an aspect by using the `Aspect Selection panel`. It is also possible for the user to choose a file for defining the variable domains from `Variable Domains File Selection panel`. Then the user is expected to choose a mode for code generation, default mode is verification mode. After that process, user can generate the base code by clicking the `Generate Aspect Environment` button. At the first tab of the detail view of the window, generated base classes are displayed as shown in Figure 56. At the second tab generated driver class, which is going to use base classes, is presented. This view is shown in Figure 57.

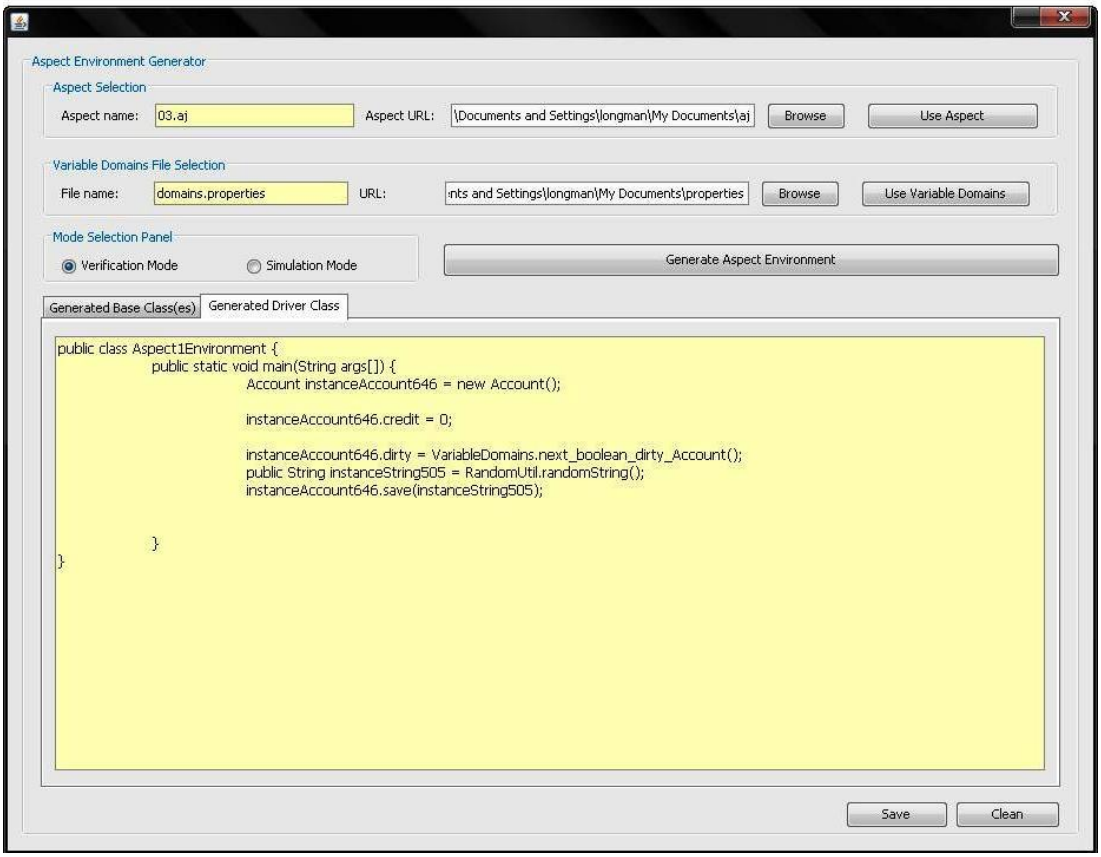


Figure 57: Environment Generator – Driver Class –

Generated classes can be saved into the file system by using the same window.

CHAPTER VI

EXPERIMENTS

In order to make some tests on the tool with different types of examples, this chapter is prepared before concluding the document of the thesis study. There are also examples that show the tricky usages of AspectJ syntax supported by the tool. Since the content of `VariableDomains` and `RandomUtil` classes are similar for all of the examples, their contents are not provided within these examples. All of the aspects that are used in the examples of this chapter are taken from [12]. To make them suitable to show the effect of our study, several updates are performed on the original aspects.

VI.1 Examples of Handling Tricky Usages of AspectJ Syntax

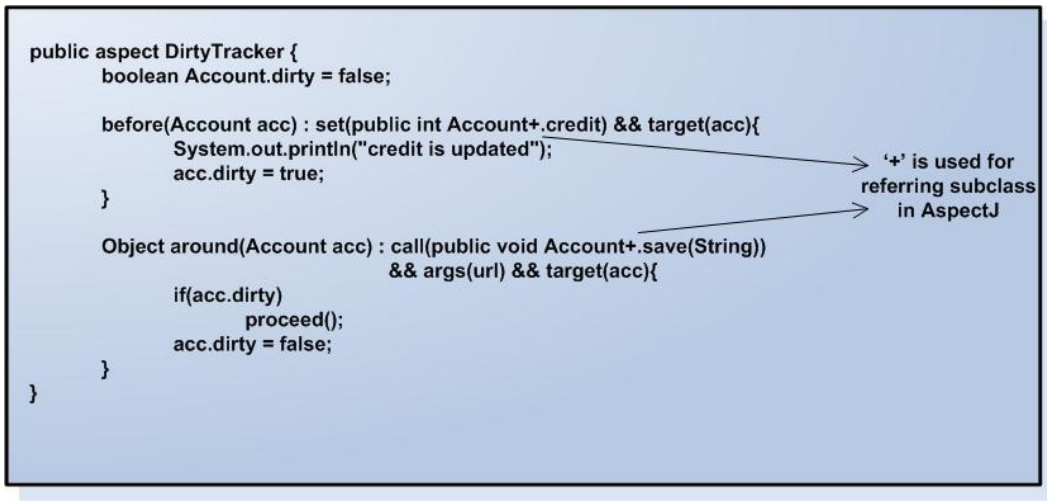


Figure 58: Usage of '+' in AspectJ

The first example focuses on the usage of `'+'` symbol inAspectJ. `'+'` is used for representing the subclasses of the class. In Figure 58, the example point-cuts catch the join points of any subclasses of `Account` class (including `Account` class itself) and our tool supports this syntax. The output of our tool for input in Figure 58 is shown in Figure 59.

```
public class Account {
        public boolean dirty = false;
        public int credit;
        public void save(String prm368) {
                dirty = VariableDomains.next_boolean_dirty_Account();

        }
}

public class Aspect1Environment {
        public static void main(String args[]) {
                Account instanceAccount155 = new Account();
                instanceAccount155.credit = 0;
                instanceAccount155.dirty = VariableDomains.next_boolean_dirty_Account();
                public String instanceString165 = RandomUtil.fixedString();
                instanceAccount155.save(instanceString165);
        }
}
```

Figure 59: Generated Environment for the Usage of '+' in AspectJ

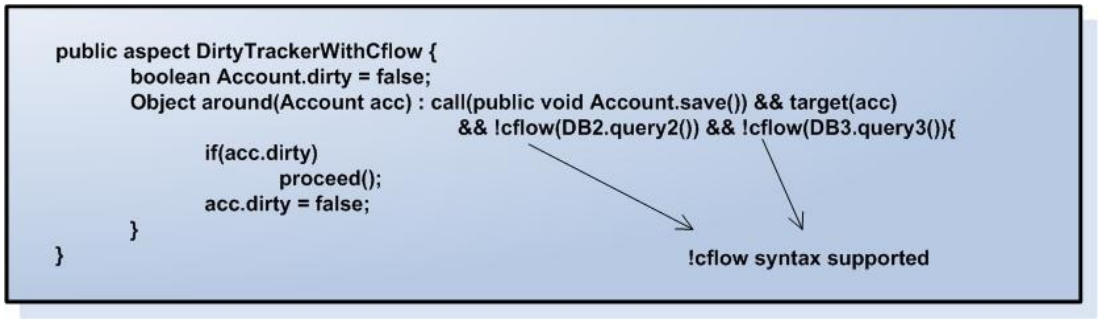


Figure 60: Usage of '!cflow' in AspectJ

Another example is about the usage of `!Cflow` keyword. `!Cflow` is a control-flow based point-cut. Differently from the point-cuts that are explained in Chapter II, join point of this point-cut is determined at run-time [12]. `!Cflow` syntax as shown in Figure 60 is supported in our tool, but `Cflow` usage is not supported. Related code generations are done in accordance with the statements in it. In other words the class and the method to generate are extracted from it. The output is represented in Figure 61.

```
public class Account {
        public boolean dirty = false;
        public void save() {
                dirty = VariableDomains.next_boolean_dirty_Account();
        }
}

public class DB2 {
        public void query2() {

        }
}

public class DB3 {
        public void query3() {

        }
}

public class Aspect1Environment {
        public static void main(String args[]) {
                Account instanceAccount360 = new Account();
                instanceAccount360.dirty = VariableDomains.next_boolean_dirty_Account();
                instanceAccount360.save();
        }
}
```

Figure 61: Generated Environment for the Usage of '!cflow' in AspectJ

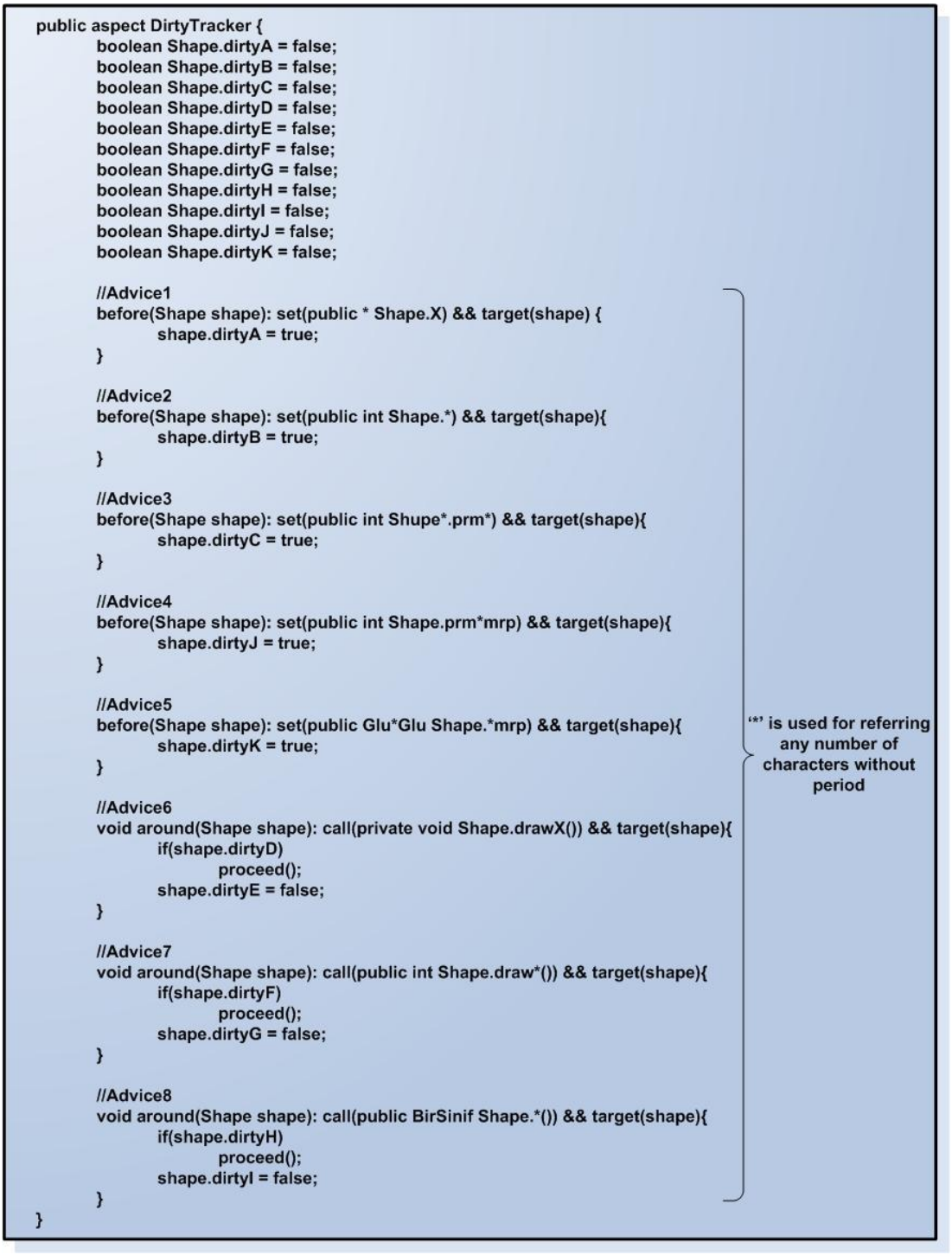


Figure 62: Usage of ‘*’ in AspectJ

Another symbol that is used in AspectJ is '*'. It is used for representing any number of characters [6]. Several examples showing the usage of this symbol are represented in Figure 62. The aim of usage of this symbol is shortcut specifying, for

example, all the attributes of an object as in Advice2 in Figure 62 .The generated environment for the aspect in Figure 62 is displayed in Figure 63.

```
public class Shape {
	public boolean dirtyA = false;
	public boolean dirtyB = false;
	public boolean dirtyC = false;
	public boolean dirtyD = false;
	public boolean dirtyE = false;
	public boolean dirtyF = false;
	public boolean dirtyG = false;
	public boolean dirtyH = false;
	public boolean dirtyI = false;
	public boolean dirtyJ = false;
	public boolean dirtyK = false;
	public boolean X;
	public int Any;
	public int prmAnymrp;
	public GluAnyGlu Anymrp;

	public void drawX() {
		dirtyD = VariableDomains.next_boolean_dirtyD_Shape();
	}

	public int drawAny() {
		dirtyF = VariableDomains.next_boolean_dirtyF_Shape();
		return RandomUtil.fixedInt();
	}

	public BirSinif instanceBirSinif769 = new BirSinif();
	public BirSinif Any() {
		dirtyH = VariableDomains.next_boolean_dirtyH_Shape();
		if(RandomUtil.randomBoolean())
			return null;
		else
			return instanceBirSinif769;
	}
}

public class GluAnyGlu {
}

public class BirSinif {
}

public class ShupeAny {
	public int prmAny;
}

public class Aspect1Environment {
	public static void main(String args[]) {
		Shape instanceShape638 = new Shape();
		instanceShape638.X = true;
		instanceShape638.Any = 0;
		instanceShape638.prmAnymrp = 0;
		instanceShape638.dirtyD = VariableDomains.next_boolean_dirtyD_Shape();
		instanceShape638.drawX();
		instanceShape638.dirtyF = VariableDomains.next_boolean_dirtyF_Shape();
		instanceShape638.drawAny();
		instanceShape638.dirtyH = VariableDomains.next_boolean_dirtyH_Shape();
		instanceShape638.Any();
		ShupeAny instanceShupeAny278 = new ShupeAny();
		instanceShupeAny278.prmAny = 0;
	}
}
```

Figure 63: Generated Environment for the Usage of '*' in AspectJ

```
public aspect DirtyTracker {
        boolean Account.dirty = false;
        int Account.cnt = 1;
        int Account.ii = 4;
        double Account.dd = 6.0;

        Object around(Account acc) : call(public void Account.save()) && target(acc){
                while(acc.cnt < 10) {
                        if(acc.dirty) {
                                for(;acc.ii < 10;) {
                                        double ydd = acc.dd;
                                }
                        }
                }
        }
}
```

conditional statements can be handled

Figure 64: Usage of Conditional Statements in AspectJ

Finally, conditional statements as in Figure 64 are supported by our tool. If there are too many conditional statements in the aspect code, it becomes very tedious to prepare its environment manually; however independently from the number of conditional statements, its environment can be generated by our tool easily. The environment in Figure 65 is generated for the given aspect.

```
public class Account {
        public boolean dirty = false;
        public int cnt = 1;
        public int ii = 4;
        public double dd = 6.0;
        public void save() {
                cnt = VariableDomains.next_int_cnt_Account();
                dirty = VariableDomains.next_boolean_dirty_Account();
                ii = VariableDomains.next_int_ii_Account();
                dd = VariableDomains.next_double_dd_Account();
        }
}

public class Aspect1Environment {
        public static void main(String args[]) {
                Account instanceAccount625 = new Account();
                instanceAccount625.cnt = VariableDomains.next_int_cnt_Account();
                instanceAccount625.dirty = VariableDomains.next_boolean_dirty_Account();
                instanceAccount625.ii = VariableDomains.next_int_ii_Account();
                instanceAccount625.dd = VariableDomains.next_double_dd_Account();
                instanceAccount625.save();
        }
}
```

Figure 65: Generated Environment for the Usage of Conditional Statements in AspectJ

VI.2 Case Study

We performed a case study on two realistic examples to show that our tool is a useful tool for AspectJ developers. The first example is a slightly different version of its original that is represented in [12]. It is a database connection pooling example and details about the problem and its solution by using AspectJ are explained in [12].

```
import java.sql.*;
public aspect DBConnectionPoolingAspect {
   public static DBConnectionPool DriverManager.connPool = new SimpleDBConnectionPool();

   pointcut connectionCreation(String url, String username, String password)
:call(public static Connection DriverManager.getConnection(String, String, String))
&& args(url, username, password);

            pointcut connectionRelease(Connection connection)
            :call(public void Connection.close()) && target(connection);

              //Advice 1
              Connection around(String url, String userName, String password) throws SQLException :
              connectionCreation(url, userName, password) {
                 Connection connection = DriverManager.connPool.getConnection(url, userName, password);
                 if (connection == null) {
                    connection = proceed(url, userName, password);
                    DriverManager.connPool.registerConnection(connection, url, userName, password);
                 }
                 return connection;
              }

              //Advice 2
              void around(Connection connection) : connectionRelease(connection) {
                 if (!DriverManager.connPool.putConnection(connection)) {
                    proceed(connection);
                 }
              }
}
```

Figure 66: A Realistic DBConnection example in AspectJ

Our focus is on the generation of the environment of the aspect in Figure 66. Base code for a realistic DBConnection aspect as in Figure 66 can be generated as well by using the tool that we have developed. The generated base code output of the tool is shown in Figure 67.

```
public class SimpleDBConnectionPool extends DBConnectionPool {
        public static boolean instanceuseless157 = true;
}

public class DBConnectionPool {
        public Connection instanceConnection308 = new Connection();
        public Connection getConnection(String url,String userName,String password) {
                if(RandomUtil.randomBoolean())
                        return null;
                else
                        return instanceConnection308;
        }
        public void registerConnection(Connection connection,String url,String userName,String password) {

        }
        public boolean putConnection(Connection connection) {
                        return RandomUtil.fixedBoolean();
        }
}

public class Connection {

        public void close() {

        }
}

public class DriverManager {
        public static DBConnectionPool connPool = new SimpleDBConnectionPool();
        public static Connection instanceConnection641 = new Connection();
        public static Connection getConnection(String prm1947,String prm9954,String prm8542) {
                if(RandomUtil.randomBoolean())
                        return null;
                else
                        return instanceConnection641;
        }
}

public class Aspect1Environment {
        public static void main(String args[]) {
                Connection instanceConnection653 = new Connection();
                instanceConnection653.close();
                public String instanceString722 = RandomUtil.fixedString();
                public String instanceString406 = RandomUtil.fixedString();
                public String instanceString764 = RandomUtil.fixedString();
                DriverManager.getConnection(instanceString722,instanceString406,instanceString764);
        }
}
```

Figure 67: Generated Environment for the Realistic DBConnection example in AspectJ

From the third line of the code in Figure 66, the tool realizes that `SimpleDBConnectionPool` is a class that is extending `DBConnectionPool` class and creates `SimpleDBConnectionPool` class according to this information. From `Advice 1` and `Advice 2` the methods of `DBConnectionPool` class are generated. From `connectionRelease` point-cut, `Connection` class and its content are genereted. From `connectionCreation` point-cut `DriverManager` class and its

content are created. The driver class `Aspect1Environment` is generated to enable the join points for verification.

```
import java.util.*;

public aspect OptimizeFactorialAspect {

   public static FactorialCache Manager.factCache = new FactorialCache();
   pointcut factorialOperation(int n) :
         call(public long Factorial.factorial(int)) && args(n);

   long around(int num) : factorialOperation(num) {
         long cachedValue = Manager.factCache.getFact(num);
         if(cachedValue == 0) {
                  cachedValue = proceed(num);
                  Manager.factCache.setFact(cachedValue, num);
                  System.out.println("Found cached value for " + num
                                    + ": " + cachedValue);
         }
         return cachedValue;
   }
}
```

Figure 68: A Realistic Factorial Optimization example in AspectJ

The second realistic example is a caching mechanism for the factorial values of the inputs of a `factorialOperation` method. The aspect is called the `OptimizeFactorialAspect` and it is provided in [12]. This aspect is shown in Figure 68, and our tool generated the environment given in Figure 69.

```
public class Manager {
        public static FactorialCache factCache = new FactorialCache();
}

public class Factorial {
        public long factorial(int prm6096) {
                return RandomUtil.fixedLong();
        }
}


public class FactorialCache {
        public long getFact(int prm3676) {
                return RandomUtil.fixedLong();
        }
        public void setFact(long cachedValue, int prm3024) {

        }
}

public class Aspect1Environment {
        public static void main(String args[]) {
                Factorial instanceFactorial854 = new Factorial();
                public int instanceint251 = RandomUtil.fixedInt();
                instanceFactorial854.factorial(instanceint251);
        }
}
```

Figure 69: Generated Environment for the Realistic Factorial Optimization example in AspectJ

A `Manager` class with its content is generated according to the static initialization line in Figure 68. The only point-cut is used for the generation of `Factorial` class and its content. `FactorialCache` and its content are generated by using the information in the advice block of the aspect. `Aspect1Environment` is generated to enable the join points for verification.

CHAPTER VII

CONCLUSION AND FUTURE WORK

In this study, we aim to describe the necessity for automation of aspect verification, and then we have proposed our solution at AspectJ programming language. Our proposed solution is a tool for generating mock classes of an aspect and making it ready for verification.

General architecture of our solution is formed of two basic parts. Firstly, the data including point-cuts, advices, variables, methods is collected from the supplied input aspect and then in the second step the output, which is the environment code of the input aspect, is generated from the collected data in the first part.

Data collection is a process that involves two sub-processes that are working subsequently. Aspect code parsing is the first one of those sub-processes. At the aspect code parsing stage, aspect code is parsed by using JTB parser [7].

The second sub-process of the data collection process is filling data objects. We traverse the output syntax tree by our visitor class that is overriding the visitor class

generated by JTB. While traversing the syntax tree we fill the data objects of classes that we prepared as suitable for code generation process.
Second step in the general structure of the project is base code generation part. At this step by using the collected information from the aspect the actual goal of this study, which is generation of base code for aspect, is completed. By using the composite design pattern [30], we have constructed a reasonable code generation algorithm. All data objects were put in another one during data filling process and from the outmost to the innermost all objects generate some part of the base code that is related with its own, and then call the method generation methods of the inner objects. By this way whole base code including the actual application classes and the trigger class is generated.

Since our aim at generating Java base code is making it possible for aspects to be verified without any other input from the user, the other significant points of verification has been considered during preparation of the base code generation methodology. The most critical one of them was the need for domain value assignments. To be sure about a complete verification, verification tools make many tries on each line of code. By making use of domain values we provide variation on code execution and become closer to generate base code that is ready for a more complete verification. Domain value assignments change according to the code generation mode selected by the user. The tool is able to generate base code in two modes; verification mode and simulation mode.

Our solution facilitates the work of a developer at aspect verification process. By taking only the aspect as input and not expecting any other effort of the developer, it proves its usefulness. The tool takes an aspect as input and produces all the necessary base classes in Java programming language. The produced output code has the following properties.

- Complete
- Correct

- General (Enabling all behavior affecting input values)
- Efficient (No unnecessary or dead code, goal oriented code)

Considering the verification process at code generation, above properties are necessary to be satisfied in the generated code.

One of the goals of this study is to discriminate the faults of aspect code from the faults of base code. By producing a complete, correct, general (enabling all behavior affecting input values) and efficient environment code (no unnecessary or dead code, goal oriented code) we are promising that the faults that developer comes across during the verification process completely belong to the aspect code.

The points that differentiate our study from the current literature and the contributions of this study are listed as follows.

- We enable modular verification and testing of the aspects on the contrary to most of the approaches that are using base code in the same processes. This modular approach that we use gives advantage of aspect reusability
- We put together both the ability of aspect environment generation for verification and ability of aspect mocking framework for unit testing in one tool. Other studies focus on just one of these purposes
- We provide a useful and detailed implementation in addition to the description of our approach
- In verification mode we consider the state space explosion problem of software verification processes and apply an original methodology for solving that problem in generated environment
- Lastly, the idea that we use for detecting aspect behavior affecting points and generating over-approximations of those points make our approach original and different from others

There are limitations for the syntax of the aspects that can be used as input for our application. The following limitations should be considered.

- The tool facilitates the use of the common point-cuts, which are method calling, method execution, and attribute setting and attribute getting point-cuts. However other types of point-cuts are not supported.
- All types of advices are supported.
- Base methods, base variables, intertype variables are supported, on the other hand intertype methods are not supported.
- The '*' (any number of any characters) and '+' (sub-classing) wild-cards can be used in point-cut declarations; other special symbols are not supported
- '!cflow' control flow point-cut can be used, but 'cflow' is not supported

If these syntactic limitations are exceeded, no output is generated by the tool. In addition to the limitations of syntax, there are other limitations for the tool usage and they are listed as follows.

- The tool should not be used in verification mode without a variable domain file.
- For random value assignments of primitive types a generator is used. However for composite types the methodology is different. For composite types there are two possibilities; one of them is assignment of a null value and the other one is an initialization with the default constructor. There is not any other methodology for random value assignments of composite types.
- Domain value assignments are similar to the random value assignments. In other words, only the domain values of primitive types are expected to be put in variable domains input file.
- There are no random value and domain value assignments for containers like arrays, vectors, and so on.
- The tool can be used for only one aspect at a time.
- Considering that the environment for an aspect is generated automatically, it is not going to be reasonable to use our application for an aspect that is checking the state of the base program. It is possible that, while the state of

the original base code was changing at any execution process, it is not going change in the process of automatically generated base code.

On the contrary to the syntactic limitations, these limitations do not prevent the generation of outputs. However, the outputs become incorrect. Before using the tool all of these limitations should be considered.

The immediate future work of this study is making our application available for global public usage over the web. By this way we aim to obtain feedback from users and consider possible improvements in the application.

Certainly the best idea as a future work of this study should be on the verification subject. The materials for verification are ready after our study. The aspect that is supplied by the developer and the base code that is generated by our tool can be used as inputs of that study. After taking the two inputs, which are aspect and base code, the application of the assumed future work is going to guide these inputs to a verification tool such as JPF, and then it is going to make some analysis on the returned result from verification tool. The difficult part of this study would be obviously on the analysis part. After the analysis is completed the application is going to display an easy to understand, useful, detailed view for the results of the analysis of verification.

As another future work of this study, it is possible to look at picture from another angle. In our study we have focused on the verification of the aspect part of an AspectJ program and generated the base code with our tool for discriminating the faults of aspect from faults of base code. Looking from the other view, it is also a good idea to find the faults of base code separately from the faults of aspect. By combining such a study with our study, a complete verification tool can be provided to AspectJ developers.

REFERENCES

[1] N. Ubayashi and T. Tamai, "Aspect-oriented programming with model checking," in *Proceedings of the 1st international conference on Aspect-oriented software development*, Enschede, The Netherlands, 2002, pp. 148-154.

[2] M. Storzer, "Analysis of AspectJ programs," *Aspect-Oriented Software Development*, p. 39, 2003.

[3] G. Holzmann, "Trends in software verification," *FME 2003: Formal Methods*, vol. 2805, pp. 40-50, 2003.

[4] H. Zhu, P. Hall, and J. May, "Software unit test coverage and adequacy," *ACM Computing Surveys (CSUR)*, vol. 29, pp. 366-427, 1997.

[5] *JPF Web site*. Available: http://babelfish.arc.nasa.gov/trac/jpf

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," *ECOOP 2001--Object-Oriented Programming*, vol. 2072, pp. 327-354, 2001.

[7] *Java Tree Builder (JTB) Web site*. Available: http://compilers.cs.ucla.edu/jtb

[8] C. Clifton and G. Leavens, "Observers and assistants: A proposal for modular aspect-oriented reasoning," in *Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, Department of Computer Science, Iowa State University, 2002, pp. 33-44.

[9] R. Filman and D. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Workshop on Advanced Separation of Concerns*, Minneapolis, 2000, pp. 21-35.

[10] R. Alexander, J. Bieman, and A. Andrews, "Towards the systematic testing of aspect-oriented programs," *Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA. Technical Report CS-4-105*, 2004.

[11] J. Pérez, N. Ali, J. Carsí, and I. Ramos, "Designing software architectures with an aspect-oriented architecture description language," *Component-Based Software Engineering*, vol. 4063, pp. 123-138, 2006.

[12] R. Laddad, *AspectJ in action*: Manning, 2003.

[13] G. Holzmann, "The model checker SPIN," *IEEE Transactions on software engineering*, vol. 23, pp. 279-295, 1997.

[14] D. Larsson and R. Alexandersson, "Formal verification of fault tolerance aspects," in *International Symposium on Software Reliability Engineering (ISSRE)*, Los Alamitos, 2005, pp. 279-280.

[15] Y. Cheon and G. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," *ECOOP 2002--Object-Oriented Programming*, pp. 1789-1901, 2003.

[16] *JavaCC Grammar Web site*. Available: https://javacc.dev.java.net/doc/javaccgrm.html

[17] J. Zhao and M. Rinard, "Pipa: A Behavioral Interface Specification Language for Aspect," in *Proceedings of the 6th international conference on Fundamental approaches to software engineering*, Warsaw, Poland, 2003, pp. 150-165.

[18] S. Krishnamurthi, K. Fisler, and M. Greenberg, "Verifying aspect advice modularly," *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 137-146, 2004.

[19] F. Mostefaoui and J. Vachon, "Verification of Aspect-UML models using Alloy," in *Proceedings of the 10th international workshop on Aspect-oriented modeling*, Vancouver, Canada, 2007, pp. 41-48.

[20] B. Devereux, "Compositional reasoning about aspects using alternating-time logic," in *Foundations of Aspect-Oriented Languages*, 2003, pp. 40-43.

[21] S. Katz and A. Rashid, "From aspectual requirements to proof obligations for aspect-oriented systems," *architecture*, vol. 25, p. 26, 2004.

[22] S. Katz and A. Rashid, "PROBE: From Requirements and Design to Proof Obligations for Aspect-Oriented Systems," *Computing Department, Lancaster University, Lancaster COMP-002-2004*, 2004.

[23] M. Rinard, A. Salcianu, and S. Bugrara, "A classification system and analysis for aspect-oriented programs," *SIGSOFT Softw. Eng. Notes*, vol. 29, pp. 147-158, 2004.

[24] M. Qamar, A. Nadeem, M. Khan, and M. Ali, "An Automatable Framework for Formal Specification & Verification of Aspect Oriented Programs," 2008.

[25] N. Ubayashi, J. Piao, S. Shinotsuka, and T. Tamai, "Contract-Based Verification for Aspect-Oriented Refactoring," presented at the *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, 2008.

[26] M. Pérez-Toledano, A. Navasa, J. Murillo, and C. Canal, "A Safe Dynamic Adaptation Framework for Aspect-Oriented Software Development," *Journal of Universal Computer Science*, vol. 14, pp. 2212-2238, 2008.

[27] M. Bagherzadeh, H. Rajan, G. Leavens, and S. Mooney, "Translucid Contracts: Expressive Specification and Modular Verification for Aspect-oriented Interfaces," 2010.

[28] *Ptolemy with Translucid Contracts*. http://www.cs.iastate.edu/~ptolemy/contract/

[29] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu, "Automatic code generation from design patterns," *IBM Systems Journal*, vol. 35, pp. 151-171, 1996.

[30] J. Cooper, *The Design Patterns Java Companion*: NY: Addison-Wesley, 1998.

[31] *ABC Web site*. Available: http://abc.comlab.ox.ac.uk/introduction

APPENDICES

APPENDIX A. RANDOM UTILITY CLASS

```
import java.util.Random;


public class RandomUtil {

    Random generator = new Random();

    public static boolean randomBoolean() {
        return generator.nextBoolean();
    }

    public static short randomShort() {
           return (short) generator.nextInt(1000);
    }

    public static int randomInt() {
           return generator.nextInt();
    }

    public static long randomLong() {
        return generator.nextLong();
    }

    public static float randomFloat() {
        return generator.nextFloat();
    }

    public static double randomDouble() {
        return generator.nextDouble();
    }

    public static String randomString() {
        return "";
    }
```

```
    public static char randomChar() {
        return ' ';
    }

    public static boolean fixedBoolean() {
        return true;
    }

    public static short fixedShort() {
     return 0;
    }

    public static int fixedInt() {
     return 0;
    }

    public static long fixedLong() {
        return 0;
    }

    public static float fixedFloat() {
        return 0;
    }

    public static double fixedDouble() {
        return 0;
    }

    public static String fixedString() {
        return "";
    }

    public static char fixedChar() {
        return ' ';
    }
}
```

APPENDIX B. ADDITIONS TO JTB GRAMMAR

- AspectDeclaration

```
{
   ModifiersOpt() ["privileged" ModifiersOpt()] "aspect"
   <IDENTIFIER> SuperOpt() InterfacesOpt() PerClauseOpt()
   AspectBody()
}
```

- AspectBody

```
{
   "{" (AspectBodyDeclarations())* "}"
}
```

- AspectBodyDeclarations

```
{
   AspectBodyDeclaration()
(LOOKAHEAD(2)AspectBodyDeclaration())*
}
```

- AspectBodyDeclaration

```
{
   LOOKAHEAD(ClassBodyDeclaration()) ClassBodyDeclaration()
   | DeclareDeclaration()
   | LOOKAHEAD(AdviceDeclaration()) AdviceDeclaration()
   | LOOKAHEAD(InterTypeMemberDeclaration())
   InterTypeMemberDeclaration()
   | PointcutDeclaration()
}
```

- DeclareDeclaration

```
{
   "declare" (
   "parents" ":" ClassNamePatternExpr()
   ("extends"|"implements") NameList() ";"
   |  "warning" ":" PointcutExpr() ":" <STRING_LITERAL> ";"
   |  "error" ":" PointcutExpr() ":" <STRING_LITERAL> ";"
   |  "soft" ":" Type() ":" PointcutExpr() ";"
   |  "precedence" ":" ClassNamePatternExprList() ";"
)
}
```

- AdviceDeclaration

```
{
   ModifiersOpt() AdviceSpec() ThrowsOpt() ":"
PointcutExpr() MethodBody()
```

```
}
```

- AdviceSpec

```
{
   "before" "(" FormalParameterListOpt() ")"
   | "after" "(" FormalParameterListOpt() ")"
   [ ("returning" | "throwing") [ "("  [ FormalParameter()
] ")" ] ]
   | Type() "around" "(" FormalParameterListOpt() ")"
   | "void" "around" "(" FormalParameterListOpt() ")"
}
```

- InterTypeMemberDeclaration

```
{
   ModifiersOpt() (
   "void" "." <IDENTIFIER> "(" FormalParameterListOpt()
   ")"   ThrowsOpt() MethodBody()
   | LOOKAHEAD(Name())
   Name() "." "new" "(" FormalParameterListOpt() ")"
   ThrowsOpt() ConstructorBody()
   | LOOKAHEAD(Type())
   Type() Name() "." <IDENTIFIER> (
   "(" FormalParameterListOpt() ")" ThrowsOpt()
MethodBody()
   | "=" VariableInitializer() ";"
   | ";"
   )
   )
}
```

- PointcutDeclaration

```
{
   <CONTRACT>
   ModifiersOpt() "pointcut" <IDENTIFIER>
   "("FormalParameterListOpt()")" [":" PointcutExpr()] ";"
}
```

- PointcutExpr

```
{
   OrPointcutExpr() ("&&" OrPointcutExpr())*
    }
```

- OrPointcutExpr

```
{
   UnaryPointcutExpr() ("||" UnaryPointcutExpr())*
}
```

- UnaryPointcutExpr

```
{
   BasicPointcutExpr()
   | "!" UnaryPointcutExpr()
    }
```

- BasicPointcutExpr

```
{
   "(" PointcutExpr() ")"
   | "call" "(" MethodConstructorPattern() ")"
   | "execution" "(" MethodConstructorPattern() ")"
   | "initialization" "(" ConstructorPattern() ")"
   | "preinitialization" "(" ConstructorPattern() ")"
   | "staticinitialization" "(" ClassNamePatternExpr() ")"
   | "get" "(" FieldPattern() ")"
   | "set" "(" FieldPattern() ")"
   | "handler" "(" ClassNamePatternExpr() ")"
   | "adviceexecution" "(" ")"
   | "within" "(" ClassNamePatternExpr() ")"
   | "withincode" "(" MethodConstructorPattern() ")"
   | "cflow" "(" PointcutExpr() ")"
   | "cflowbelow" "(" PointcutExpr() ")"
   | "if" "(" Expression() ")"
   | "this" "(" TypeIdStar() ")"
   | "target" "(" TypeIdStar() ")"
   | "args" "(" TypeIdStarListOpt() ")"
   | Name() "(" TypeIdStarListOpt() ")"
}
```

- AspectJReservedIdentifier

```
{
    "aspect" | "privileged"
   | "adviceexecution" | "args" | "call" | "cflow" |
   "cflowbelow" | "error"
   | "execution" | "get" | "handler" | "initialization" |
   "parents"
   | "precedence" | "preinitialization" | "returning" |
   "set"
   | "soft" | "staticinitialization" | "target" |
   "throwing"
   | "warning" | "withincode"
}
```

[31]

APPENDIX C. DETAILS OF JTB SYNTAX TREE USAGE

Figure 70 is showing the branches of an example syntax tree produced by JTB from the highest level, `AspectDeclaration`, to `AdviceDeclaration`.

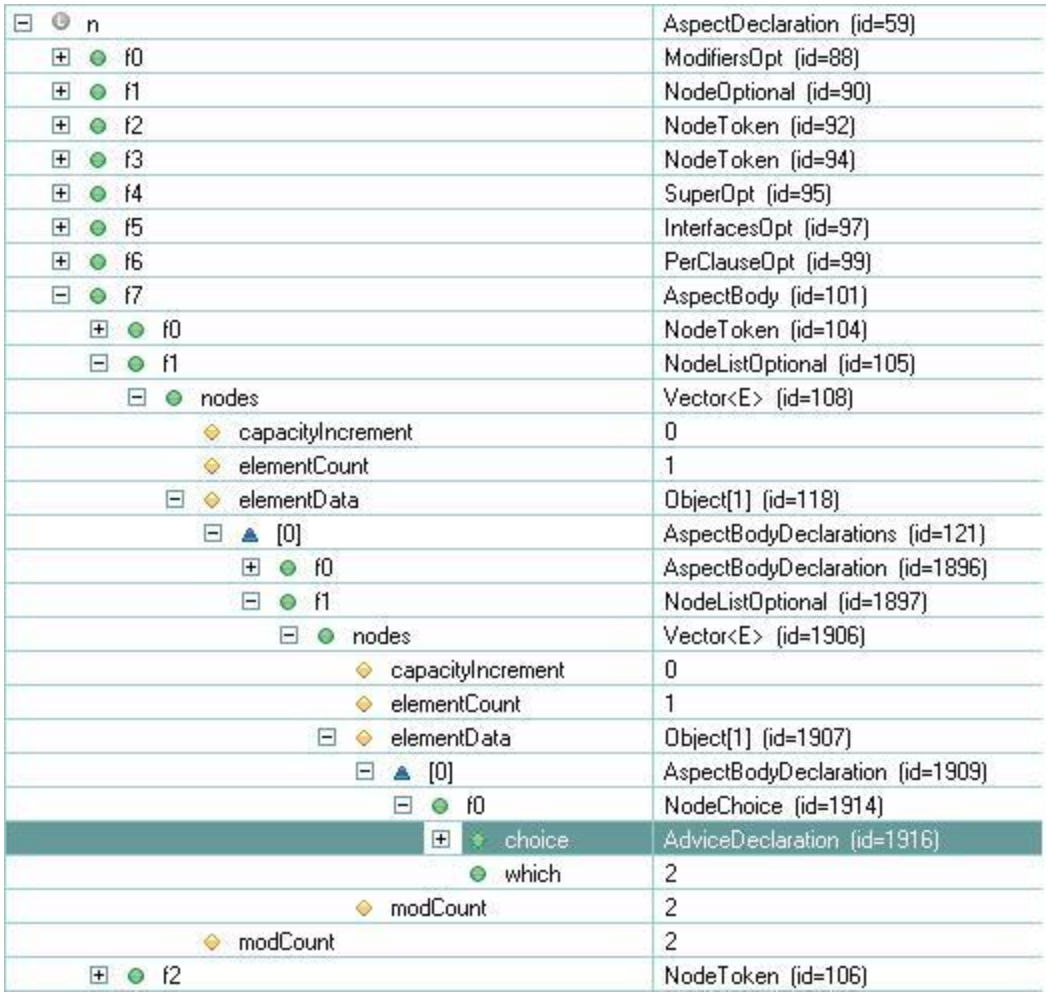


Figure 70: JTB Syntax Tree – Aspect Content – (Debug View)

In order to generate the base code from the aspect code, the necessary nodes of syntax tree need to be visited. For example, to find the `AdviceDeclaration` from the `AspectDeclaration` node, first node `f7` of `AspectDeclaration` is taken. This kind of information is fixed information and we are sure that node `f7` is of type

`AspectBody`. From `AspectBody` by visiting the shown nodes in Figure 70 we find `AspectBodyDeclaration`. Here there is a conditional case from `AspectBodyDeclaration` to `AdviceDeclaration`, because it is not certain that every `AspectBodyDeclaration` has to involve `AdviceDeclaration`. This kind of conditional cases in the program is solved as shown in the Figure 71.

```
AspectBodyDeclaration aspectBodyDecl; //assume that we are already at this level
if (aspectBodyDecl.f0.present && aspectBodyDecl.f0.which == 2) {
        AdviceDeclaration advDecl = aspectBodyDecl.f0.choice;
}
```

Figure 71: Aspect Body Declaration

If there is an `f0` node in the `aspectBodyDecl` we look for the type of data in it. `aspectBodyDecl.f0.which == 2` statement does this work. JTB syntax tree provides this information us to differentiate the types of branches.

n	AdviceDeclaration (id=64)
f0	ModifiersOpt (id=100)
f1	AdviceSpec (id=102)
f2	ThrowsOpt (id=104)
f3	NodeToken (id=106)
f4	PointcutExpr (id=108)
f5	MethodBody (id=110)

Figure 72: JTB Syntax Tree – Advice Content – (Debug View)

After we have `AdviceDecleration` in hand, we can search the necessary information that we need for base code generation. Assume that we need to know the type of advice. In other words we want to learn if it is a ‘before’, ‘after’ or ‘around’ type advice. That information is supposed to be under `AdviceSpec` sub-branch. It means that we continue our search on `f1` node of `AdviceDeclaration` as shown on Figure 72. The way to access the `before` tokenImage is shown in Figure 73. As shown here, to find the necessary information, the syntax tree is traversed.

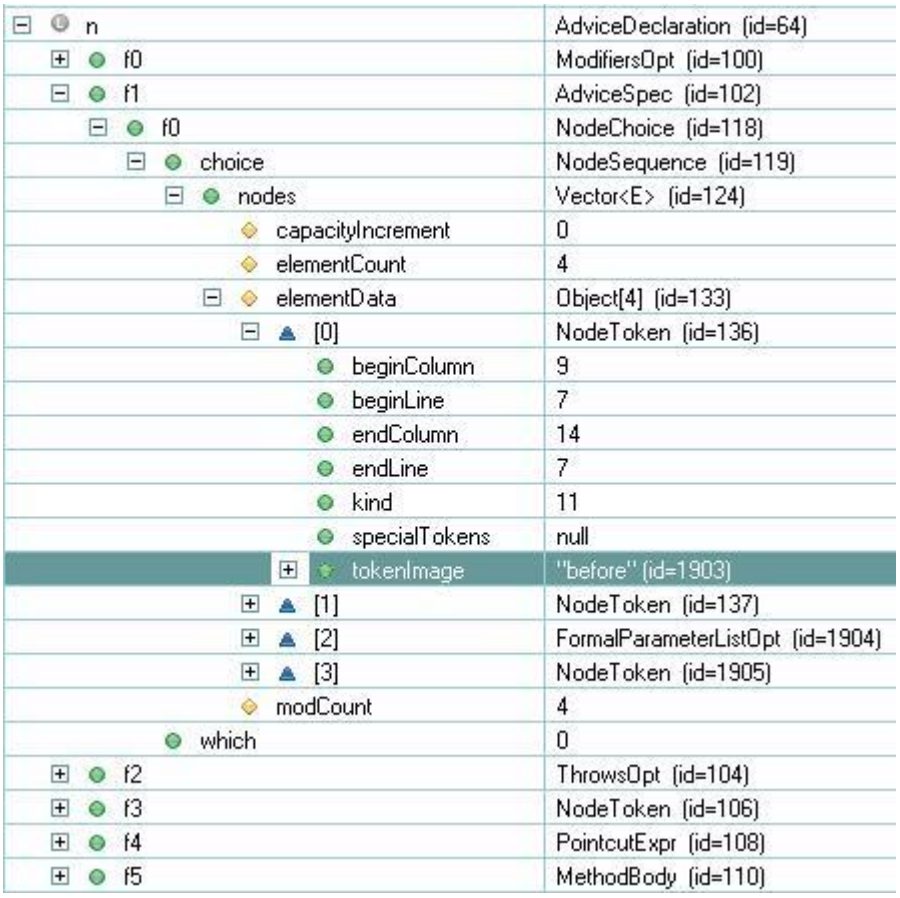


Figure 73: JTB Syntax Tree – Point-cut Type in Advice Content – (Debug View)