# NOTE TO USERS

# Applying AspectJ for Source Code Instrumentation to Support Tracing of Functional Features

Bing Chen

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

February 2006

*Your file Votre référence*
*ISBN: 978-0-494-14318-6*
*Our file Notre référence*
*ISBN: 978-0-494-14318-6*

# ABSTRACT

## Applying AspectJ for Source Code Instrumentation to Support Tracing of Functional Features

Bing Chen

Program comprehension is an essential part of software maintenance and there exists a variety of techniques the can be applied to support the comprehension process. Two of these techniques are source code analysis and program instrumentation. Source code analysis is typically applied to analyze the source code of existing systems and their behavior. There exists a wide range of techniques and approaches to support source code analysis. Feature analysis is one of these techniques that can be applied to extract functional features from existing source code. Another approach is program instrumentation that is used to collect program executions and program behavior. In this thesis, both of these techniques, feature analysis and program instrumentation, are combined to improve on the understanding of functional features and their behavior. In the presented approach, feature related statements in Java programs are mapped in a semi-automatic process into AspectJ tracing program. The approach takes advantage of both, the tracing and encapsulation support embedded within the Aspect Oriented Programming (AOP) paradigm. An initial case study is presented to illustrate the presented approach.

# ACKNOWLEDGEMENTS

I would first like to thank my supervisor, Dr. Juergen Rilling, for his encourage, support, and guidance over the past two years. Without his help, I don't know how to make the completion of my thesis possible. I am grateful to Dr. Constantinides, whose valuable feedback helped me to improve the dissertation in many ways. My thanks also go to Dr. Grogono and Dr. Harutyunyan, for helping me to review this thesis.

I would like to thank all the members of the CONCEPT research group, with whom I had useful discussions while working on this research. I am particularly indebted to my colleague Wenjun Meng, Yonggang Zhang, Susmita Haldar, for their great help on my research.

Finally, I would like to thank my parents and family for their support and encouragement. In particular, I would like to express my gratitude to my wife,
Jingyi Wen, for having patiently supporting me in my study.

## Table of Contents

## List of Figures

# 1. Introduction

Software maintenance is an important phase in the software developing life cycle. Program comprehension also plays a crucial part in software maintenance and evolution due to the time required to understand existing programs and accounts for about half of a the total maintenance cost [Sch1995].

During the comprehension of source code, one commonly used approach is to reduce the size of the source code that has to be comprehended. There exist several analysis techniques that can be applied to support the comprehension process, such as, program slicing [Wei1982], and feature analysis [Kosc2000].

Another approach to address the comprehension problem is to improve on the current programming paradigm. Over the last decade one has seen a shift in the programming language paradigm from functional, to object-oriented and more recently to aspect oriented programming [Kicz1997]. This change in the programming paradigm was driven partially by the improved tracing functional behaviors in these new programming paradigms.

In this thesis, the focus is to improve program comprehension by monitoring features extracted from object-oriented programs and tracing these features by taking advantage of some aspect-oriented programming language features.

The remainder of this thesis is organized as follows. In chapter 2, an overview of software maintenance, program comprehension, source code analysis, program instrumentation and AspectJ programming language are presented. In chapter 3, the motivations of the presented work as well as the design and implementation of this project are described. Chapter 4 presents a case study to illustrate and validate the creation and use of the AspectJ tracing program, as well as a discussion on the benefits and disadvantages of the presented approach. Chapter 5 presents the conclusions and includes the discussion about future research.

# 2. Background

## *2. 1 Software Comprehension*

Software comprehension is "a process that uses existing knowledge to acquire new knowledge" [kni1998]. It is a crucial part in software maintenance and evolution due to the time required to understand existing programs and accounts for about half of the total maintenance cost [Sch1995].

Software comprehension is categorized by the way programmers understand software systems. Comprehension models are used to depict the comprehension process that is applied to understand the systems. The following section describes the major existing comprehension models, namely the top-down model, bottom-up model, and integrated model.

### 2.1.1 Top-down Model

In the top-down model [Brok1983], hypotheses are the main driver of the comprehension process. A main hypothesis that corresponds to a high level description of the program structure is created. This hypothesis is typically based on existing domain knowledge. Supplementary hypotheses are then created to support the main hypothesis. For each program component, the programmer attempts to confirm or reject the hypothesis by examining the source code. If a hypothesis is rejected, a new hypothesis may be formed and validated again. This process continues until the programmer gains enough understanding of the program [Brok1983].

### 2.1.2 Bottom-up Model

The bottom-up model [Shnd1979] can be seen as the opposite of the top-down model. In this approach, the programmer starts the comprehension process by understanding small program fragments with low level detail that are used to identify and comprehend larger code fragments. This process continues iteratively until all program functions are discovered [Shnd1979].

During code reading, the programmer searches for critical subroutines in the program and determines their functionality. Once the functionality is identified, the knowledge is used to describe a block of code that implements the particular behavior. The programmer works through the whole program hierarchically in this manner and constructs abstraction to describe higher level components until the function of the program is determined [Shnd1979] [Stor1999].

### 2.1.3 Integrated Model

It has been shown [Stor1999] that neither the bottom-up model nor the top-down model reflects the software comprehension process during the understanding of large systems. Therefore, the integrated model combining both bottom-up model and top-down model was introduced. It is being typically used in understanding large system. In this model, people alternatively utilize bottom-up and top-down during the process of comprehension.

### 2.1.4 Comprehension Model Components

Commonly there are three key components used to describe the comprehension process [Stor1999]. These components include a knowledge base, a mental model and an assimilation process. The knowledge base is based on the programmer's expertise and background knowledge in the particular problem and programming domain. The mental model describes the integration skills and the ability of a programmer understanding of the software system. The assimilation process applies either bottom-up or top-down to describe how the mental model evolves using the programmer's knowledge base, program source code and documentation. The central activity in the process is inquiry episodes, which consists of asking a question, conjecturing and answering, and then searching through source code and documentation to support or refute the answer [Stor1999].

The above components are used to build a general model to describe how programmers comprehend existing software system. These components themselves depend on the availability of analysis, filtering and searching techniques, to facilitate the understanding of the source code. Examples for such supporting techniques are source code analysis, software visualization, program instrumentation, software refactoring, etc. In the following section, we focus on the source code analysis.

## *2.2 Source Code Analysis*

Source code analysis is used to analyze and extract the system behavior to facilitate program comprehension. Source code analysis covers a wide range of topics,

including program slicing, and feature analysis. More details are discussed in the following sections.

### 2.2.1 Program Slicing

Program slicing simplifies a program by removing parts of the program that are not relevant to a particular slicing criterion, Therefore, programmer can focus on those statements that are related to the function or variable of interest [Wei1982]. Weiser observed that a slice can be more easily understood than a complete program, due to its reduced size (number of source code statements). Over the last two decades, several program slicing approaches have been proposed [Wei1982][Kor1988] [Kor1994][Hor1988]. These approaches are classified as static program slicing and dynamic program slicing. Static program slicing, introduced by Weiser in 1982 [Wei1982], extracts program slices based on static information from the source code. This approach employs data flow equations to find statements that are relevant to a set of variables at a certain point of interest. In 1988, Korel and Laski [Kor1988] introduced dynamic program slicing, which is based on dynamic execution information.

#### 2.2.1.1 Static Slicing

Based on Weiser's definition, static slicing is defined as follows: a slice *S* is computed by deleting unrelated statements with respect to a slicing criterion *C* from a program *P*. Criterion *C* is specified by a variable *v* at certain statement *T*. Slice *S* is called a static slice of criterion *C* for variable *v* at statement *T* [Wei1982].

Static program slicing is implemented by backward slicing or forward slicing approaches. In general, a backward slice is computed backward from point of interest to identify all statements and control predicates, which are relevant for the computation of the slicing criterion. Whereas, a forward slice is computed forward to extract all statements and control predicates that are affected by the slicing criterion [Hor1988] [Har2001].

Weiser's static program slicing approach computes consecutive sets of relevant statements according to data dependence and control dependence within the source code. For example,

```
1 input(n,a);
2 max := a[1];
3 min := a[1];
4 i:= 2;
5 s:= 0;
6 while i ? n do
  begin
7 if max < a[i] then
    begin
8     max := a[i];
9     s := max;
    end;
10 if min > a[i] then
  begin
11  min := a[i];
12  s := min;
  end;
13 output(s);
14 i:= i+2;
 end;
15 output(max);
16 output(min);
```

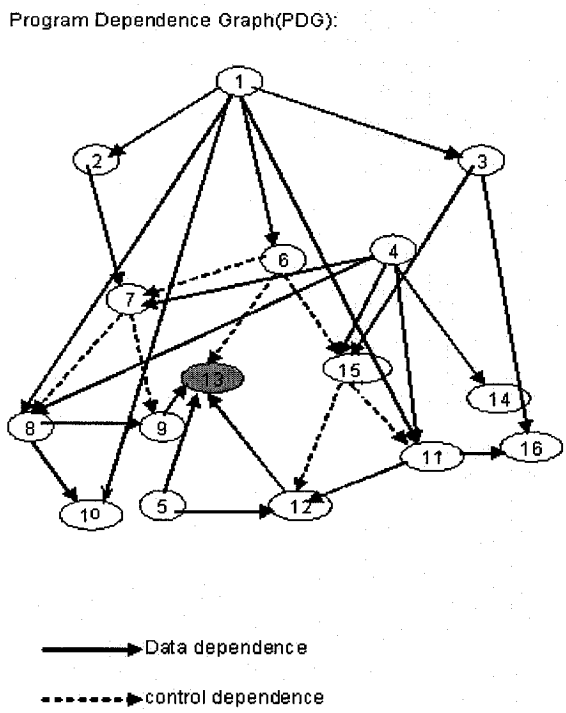



**Figure 2.2.1** Sample program PDG

Figure 2.2.1 shows a sample program and its corresponding PDG (Program Dependence Graph) that is based on data dependence and control dependence. In this example, variable *s* at output statement 13 has data dependencies with statement 12, 9, 5, and it also has a control dependency with statement 6. Therefore, the static slice of variable *s* at statement 13 consists of statement 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, and 13.

### 2.2.1.2 Dynamic Slicing

Dynamic program slicing, introduced by Korel and Laski in 1988 [Kor1988], uses dynamic information that is derived from program execution based on particular input. Dynamic slicing can therefore be seen as a refinement of the static slicing approach by utilizing additional execution information on some specific program inputs.

Based on Korel and Laski's definition[Kor1988], a dynamic slice is defined as follows: a dynamic slicing criterion of program $P$ executed on input $x$ is a tuple: $C=<x, y_q>$, where $y_q$ is a variable $y$ at execution position $q$. An executable dynamic slice of program $P$ on slicing criterion $C$ is any syntactically correct and executable program $P'$. $P'$ is obtained from $P$ by deleting zero or more statements. When program input $x$ produces an execution trace $T'_x$ for which there exists the corresponding execution position $q'$, the value of $y_q$ in $T_x$ equals the value of $y_q'$ in $T'_x$. A dynamic slice $P'$ preserves the value of $y$ for a given program input $x$ [Kor1988].

In dynamic program slicing, only the dependences that occur in a specific execution of the program are taken into account. Ideally, a dynamic slice is an executable part of a program *P,* whose behavior is identical for the same program input with fixed values. Only the subsets of the source code that occur in a specific execution of the program are taken into account [Kor1988].

Program slicing can further be divided into backward slicing and forward slicing. Backward algorithms trace backward [Kor1988] to derive data and control dependencies from a recorded execution trace to compute a dynamic slice. In contrast, forward algorithms [Kor1994] compute a dynamic slice during the execution of the program, so it does not require recording the execution trace. Forward slicing computes dynamic slices for all variables at run-time, while backward approach computes a slice only for a particular variable.

### 2.2.3 Feature Analysis

In software engineering, software systems are designed based on modules that are interacting with each other but are clearly separated (decoupled). These modules encapsulate source code that implements a particular functionality of the system. Such loosely coupled modules can be manipulated almost independently from other modules, allowing for distributed development, improved comprehensibility, and additional flexibility compared to strongly coupled systems [Thom2001] [Kosc2000].

Any coherent and identifiable bundle of system functionality perceived by users may be viewed as a *feature* of the system. Users request new functionality and report defects in terms of features. In fact, researchers suggest regarding features as a kind of objects within the software development, so it bridges the gap between the user needs and design or implementation [Kosc2000].

Typically, the implementation of a given feature is typically scattered across several software artifacts. This leads to a design that every single artifact often contains statements related to more than one feature, and several features are tangled in the same source code artifact. The major challenge of feature analysis is to comprehend how a certain feature is implemented, and distributed among the different program artifacts. The major challenge in feature comprehension at the source code level is the fact that it requires localizing the scattered implementation of the feature in the source code.

Typically, this process is even further complicated by the fact that existing documentation is outdated, the system's original architects are no longer available or their views are outdated due to changes made by others [Kosc2000].

There exist several different feature analysis approaches to guide the feature comprehension process at the source code level. Feature analysis considers certain functionality of the program as a feature, and analyzes the feature by the methodologies suitable for this functionality. Feature analysis can be divided into static and dynamic feature analysis. Static feature analysis facilitates constant

functionality, such as a method of one class; while dynamic feature analysis provides better performance for analyzing a specific executable functionality, such as certain variable's change during a program execution.

### 2.2.3.1 Static Feature Analysis

Chen and Rajlich [Chen2000] proposed a semi-automatic method for feature localization, in which an analyst browses the statically derived system dependency graph (SDG) to extract the desired feature.

The SDG describes detailed dependencies among subprograms, objects, classes, and variables at the level of individual expressions and statements. An SDG example is shown in Figure 2.2.2:

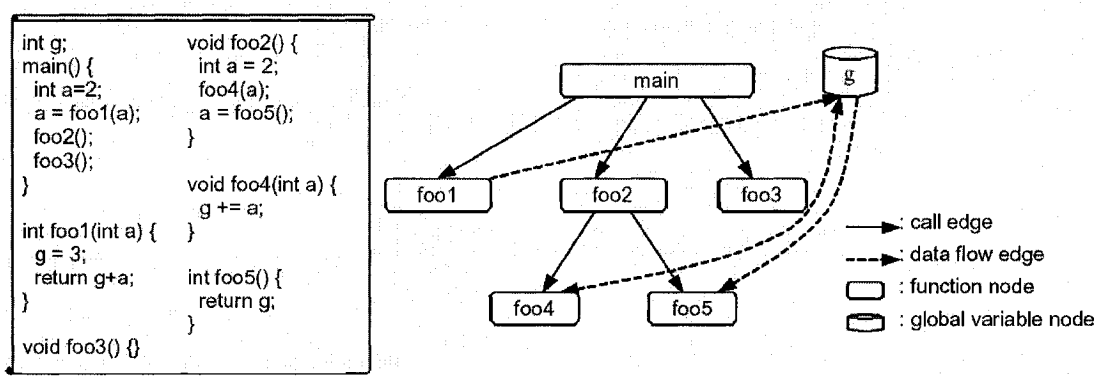



**Figure 2.2.2** SDG describes feature relationship [Chen2000]

In the SDG (Figure 2.2.2), foo4 has control dependence with foo2 and data dependence with variable g, because foo2 has control dependence with main, foo4 has control dependence with both foo2 and main. Therefore, if foo4 is the feature to

be analyzed, then it has data dependence with variable g, and control dependence with main and foo2.

Even though the navigation through a SDG is computer-aided, it is not easy for people to analyze all the information of a feature's implementation. Thus, this method is less suited to locate features quickly and cheaply if it starts without any pre-knowledge on where to begin searching.

Moreover, the method relies on the quality of the SDG. If the SDG includes overoptimistic assumptions on function pointers, the analyst may miss functions called via function pointers. If it reflects too conservative assumptions, the search space increases drastically. This is caused by uncertainty of which control flow path is taken at runtime, so every conservative static analysis yields an overestimated search space. In contrast, dynamic analysis exactly tells which parts are really used at runtime for a particular run.

#### 2.2.3.2 Dynamic Feature Analysis

Wilde and Scully [Wild1995] use a dynamic analysis to localize features in source code. Their approach is described in below:

1. The *invoking input set I* (i.e., a set of test cases) is identified that will invoke a feature.
2. The *excluding input set E* is identified that will not invoke a feature.
3. The program is executed twice using *I* and *E* separately.
4. By comparing the two resulting execution traces, the subprograms that implement input feature are identified.

Dynamic feature analysis localizes the feature related components in source code by comparing traces between invoking input set and excluding input set. Suppose we want to analyze the feature of display-update in an editor program, several tests, including a test with the feature and several tests without exercising the feature, need to be run to get the different execution traces. The components of display-update feature are located by taking the trace of the display-update test and subtracting traces of those that do not involve the display-update feature. This approach is typically not sufficient to identify the interface and the constituents of a feature in the source code, but they can be applied as starting points for further, more detailed analysis.

### 2.2.4 Slicing Based Feature Analysis

In general, source code based feature analysis describes the process of identifying all statements in source code that contribute to the implementation of a particular feature in software system.

One approach to identify and extract features from an existing source code is to apply program slicing as the underlying analysis technique. In this particular application of program slicing, it is assumed that a feature corresponds to all program input and output statements. Feature extraction based on program slicing [Sus2005] requires the identification of some input and output features. Input features are regarded as any statement in the source code that either requires an external program input (e.g. user input, input created by reading from a database, etc.). Similarly output features correspond to statements that output program

information either to the user, file or any other media. The approach is depicted in Figure 2.2.3:

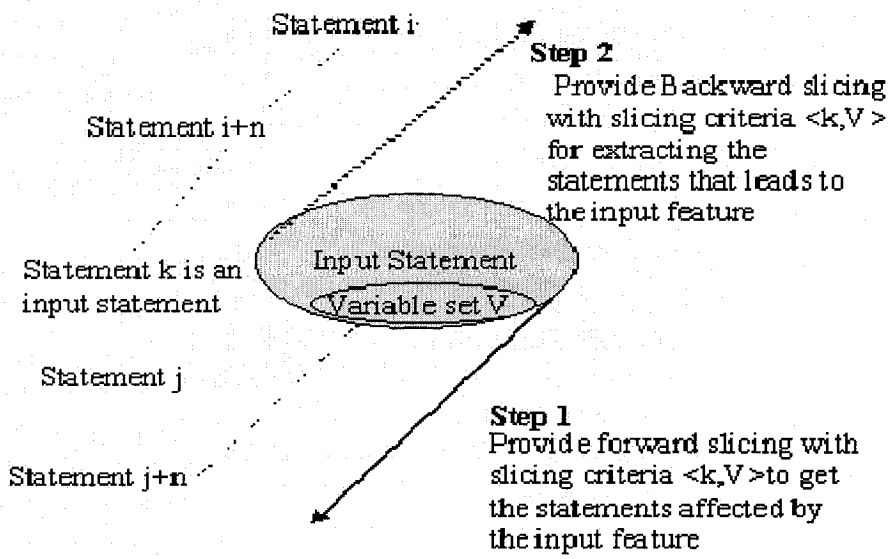



**Figure 2.2.3** Slicing based feature analysis [Sus2005]

In Figure 2.2.3, the slicing criterion of this input feature is an input statement, and forward slicing is applied to identify the potentially affected statements. After performing forward slicing, some parts of slicing are still missing, such as initialization of the variables, so backward slicing is used to identify those statements. Hence, an input feature slice is derived through a combination of forward and backward slicing.

For instance, if a user wants to withdraw money from an ATM machine, the machine will ask the user to enter the amount to be withdrawn. Based on this input amount, the forward slicing on the withdrawal operation will be performed. In this case, the slicing criterion becomes output statement that leads to the input feature, and backward slicing is applied to identify the statements that lead to the output statement [Sus2005].

## 2.3 Program Instrumentation

Program instrumentation is an important approach to trace and collect information relevant for different types of source code analysis [Ham2004]. Program instrumentation can be described as "the act of injecting monitoring code into the software system that is the subject of the monitor" [Mahr2002]. The collected information from program instrumentation can be used to analyze source code, such as profiling, generating execution trace, monitoring, testing tools, etc. There exist two instrumentation techniques in Java applications. The first technique is source code instrumentation. The second one is byte code instrumentation. More details are discussed below.

### 2.3.1 Source Code Instrumentation

Source code instrumentation inserts break points directly into the original source code. After that, the modified source code needs to be recompiled in order to run the instrumented application and obtain dynamic trace information. Several source code instrumentation tools and techniques are introduced and described briefly below:

The first example for a source code instrumentation tool is *Clover*, which is a commercial software tool developed by Cenqua Pty Ltd [Clo2005]. *Clover* utilizes source code instrumentation for code coverage analysis. It gathers and records the part of program that relates to the points of interest, which can be statements, methods, or branches. *Clover* recompiles and runs instrumented source code in a standard Java environment to derive the execution trace. In addition, *Clover* provides

fully integrated plug-ins for many popular development environments, such as *Eclipse*, *NetBeans*, *JBuilder* and *Jdevelope* [Clo2005].

The second source code instrumentation tool is *Daikon. Daikon* is a dynamic invariant detector tool developed by the Program Analysis Group in Massachusetts Institute of Technology [Dai2005]. An invariant is "a property that holds at a certain point or points in a program" [Dai2005]. *Daikon* monitors and reports the properties of the invariants during the execution of the program. *Daikon* can be used in system that developed with Java, Perl, C, C++, and it provides a plug-in for *eclipse* developing environment [Dai2005].

The third tool is *MetaC source code instrumentation* developed by Thomas Maier et al [Mai2005]. In this tool, basic code blocks are automatically determined and break points are inserted at the boundaries of these blocks. This tool is developed with MetaC programming language (a special extension of C language), and it is used to provide measurement for C software system [Mai2005].

The advantage of source code instrumentation is that it does not require any specific runtime environment. The instrumented source code is recompiled and executed within the same environments as the original programs, i.e. the same JVM and the same class loader. Another advantage is that it fits the requirements of source code analysis by providing information at different levels of abstraction. For example, *Clover* can instrument and collect information in the source code at the statement, method, or branch level [Clo2005].

### 2.3.2 Byte Code Instrumentation

The *Java Virtual Machine* (JVM) is an abstract computing machine that is implemented by simulating it in software on a real machine [Lin1999]. As the crucial component of the Java platform, JVM provides hardware independence and operating system independence for Java platform. The JVM has an instruction set, known as JVM byte code [Lin1999]. The Java source code is normally compiled to a byte code file (i.e. the class file) by JVM. Therefore, the Java byte code file is both hardware and operating system independent.

The instrumentation of Java byte code generally inserts a specific sequence of byte code at the designated locations in the compiled Java byte code file. One byte code instrumentation tool is *BIT (Bytecode Instrumentation Tool)* that was developed by H. B. Lee, and B. G. Zorn [Lee1997]. *BIT* inserts break points at specific locations in the byte code file, such as, before and after a method, before and after a basic block, or before and after one instruction. When the instrumented program runs under JVM, the trace information is collected, while the inserted break points have no impact on the original source code. *BIT* is used to rearrange procedures and to reorganize data stored in Java class files.

The Java byte code instrumentation allows for a dynamic analysis of those instrumented classes, for debugging, tracing, testing, profiling, and monitoring purpose. The advantage of Java byte code instrumentation is that it does not require a modification of the original source code.

## 2.4 Reverse Engineering, Program Migration, and Software Refactoring

In this section, a brief literature review on reverse engineering, program migration, and software refactoring is presented. These techniques are frequently used to either support or improve program comprehension.

### 2.4.1 Software Reverse Engineering

As Michael L. Nelson [Mich1996] defined, "Reverse engineering is the process of identifying software components, their interrelationships, and representing these entities at a higher level of abstraction". The aim of reverse engineering is to extract specification from the source code.

Examples of reverse engineering include: decompilation in which an object program is translated into a high level program, architecture extraction in which the design of a program is derived, documentation generation, and software visualization in which some aspects of a program is depicted in an abstract way.

Reverse engineering can be classified in four levels [Mich1996]:

- *Redocumentation:* Recovers documentation from the source code as the original documentation might be inconsistent, incorrect, or not existent.
- *Design Rediscovery:* Rediscover the design at a higher level of abstraction.

- *Restructuring:* Convert the system into another form, but keep the functionality and external behavior of the system.
- *Reengineering:* Redesign the whole or part of software system due to the updated requirements.

There exist several reverse engineering approaches, which are categorized in below [Mich1996]:

- *Syntactic analysis* - the approach takes advantage of syntax metrics and automatic parsing to search for cliches, which are extracted from the source code to give suggestions about design decisions.
- *Graphing methods* - the approach uses graphing method to implement reverse engineering task. There are a variety of graphing approaches, which include graphing the control flow of program, the data flow of program, and program dependence graphs. The unit of examination is a graphical representation of the program.
- *Execution and testing* - using execution and testing to implement reverse engineering task. There are a variety of methods for profiling, testing, and observing program behavior, including actual execution, dynamic testing, dynamic debugging, and inspection walkthroughs. The unit of examination is a full, or partial, or simulated execution of the program.

### 2.4.2 Program Transformation, Translation and Migration

#### 2.4.2.1 Program Transformation

Program transformation is "changing one program into another" [Van2002]. Based on the differences of the applied programming language, program transformation is divided into two major categories. If the source code language and destined code language are different, it is called software translation; otherwise the process is often referred to software rephrasing. In addition, these two categories are further divided into sub categories in below [Van2002]:

**Translation**

- Program Migration
- Program Synthesis
- Reverse Engineering
- Program Analysis

**Rephrasing**

- Program Normalization
- Program Optimization
- Program Refactoring
- Program Reflection
- Reengineering

Since the focus of this thesis is software translation, reverse engineering and software migration, more details about these techniques are covered in the following sections.

### 2.4.2.2 Software Translation

As described before, software translation is the process of translating a program from one programming language into another programming language. Moreover, software translation is classified into several categories by the level of abstraction of the program. One important characteristic of software translation is that some internal behavior and external behavior of the system are modified in the process. The reason is that the different programming languages or environments lead to different implementation of the system. These subcategories of software translation are depicted in Figure 2.4.1 [Van2002]:

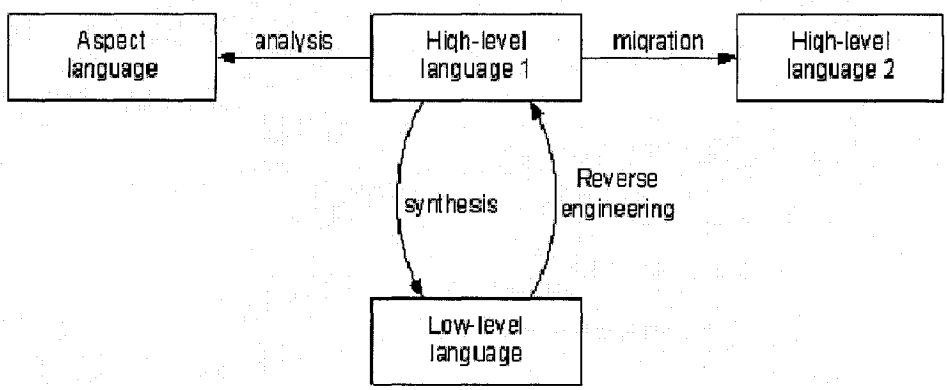



**Figure 2.4.1** Software translation category [Van2002]

### 2.4.2.3 Program Migration

In program migration, a program is transformed from one programming language into another. The migration happens between different levels of the same programming language, such as transforming a system from C++ 2.0 to VC++5.0, or

converts a system into a different programming language, e.g. converting a Foxbase program into a Visual Basic program.

Approaches for the above two program migrations are quite different. In the first situation, programmers keep programming in the same programming language, follow the design, document, and source code, and modify the program as little as possible. Secondly, in converting from one programming language to another one, programmers throw away the source code, just follow the specification, and modify system depending on the new language [Andr2001].

#### 2.4.2.4 Software Refactoring

As Ralph E. Johnson defined, "software refactoring is a program transformation that reorganizes the program without changing its behaviors. A refactory is language independent, and make program reusable and easier to maintain" [Ralph1993]. Martin Fowler and Kent Beck have used the word "code smells" to describe "bad design" [Mart1999] [Wake2002]. These "code smell" include [Mart1999]:

1. Classes that are too long
2. Methods that are too long
3. Switch statements (instead of inheritance)
4. "Struct" classes (with getters and setters but not much functionality)
5. Duplicate code
6. Almost duplicate code

7. Over-dependence on primitive types (instead of introducing a domain-specific type)
8. Too much string addition
9. Useless comments

As Martin Fowler [Mart1999] described, software refactoring typically consists of many small steps. In each step, the system behavior does not change its functionality. Programmers typically make bug fixes and feature additions in these small steps. Therefore, refactoring is a different behavior from reconstructing code.

The benefits of software refactoring are:

- easier to rearrange the code correctly if its functionality does not change
- easier to change functionality when the code is clean

Practically, the aim of refactoring is to make source code clearer, simpler, and more elegant than before. At the same time, refactoring strategies must preserve the behaviour of the system. Therefore, the major property of software refactoring is to maintain the external behaviour of the source code.

Over all, the purpose of software refactoring is to facilitate software comprehension by improving the internal structure of software and maintain the system's external behavior at the same time. In order to preserve the functionality, software refactoring process is divided into many small steps, each step comes with software testing to protect the original functionality.

### 2.4.3 Discussion on Reverse Engineering, Program Migration, and Software Refactoring

These three technologies are important techniques with differences in their purposes. Reverse engineering is mainly concerned with the comprehension of source code without modifying the original source code. Reverse engineering is an important first step for many program comprehension tools. Program migration focuses on migrating source code into another programming language while keeping system functionality. Software refactoring focuses on improving software quality. Software refactoring is quite similar to program migration in which they both keep the external behavior of the system. However, software refactoring rewrites the source code in the same programming language, rather than program migration translating the source code into another programming language.

## 2.5 Aspect Oriented Programming and AspectJ

### 2.5.1 Overview of AOP and AspectJ Language

Aspect-oriented programming (AOP) provides mechanisms that allow for the modularization and implementation of *crosscutting concerns* [Kicz1997].

When two or more functions cut across others, which leads to complexity and mix-up in source code, this situation is called *crosscutting concern*. Typical *crosscutting concerns* include error and failure handling, and performance optimization. AOP supports the encapsulating the *crosscutting concern* as module unit ***aspect,*** and AOP also provides a compiler mechanism, *aspect weaver*, to link the original program and *aspect* part [Kicz1997].

**AspectJ**

AspectJ is a Java specific implementation of *Aspect* oriented programming [Greg2001]. It was originally introduced by Xerox Palo Alto Research Center in 1997 [Kicz1997].

There are three major components in *Aspect* oriented programming [Greg2001]:

- *Join Point*
- *Point Cut*
- *Advice*

***Join Point*** "provides the common frame of reference that makes it possible for execution of a program's aspect and non-aspect code to be properly coordinated" [Greg2001]. Some typical *join points* in AspectJ are [Greg2001]:

- *method call/constructor call:*
  A method or a constructor of a class is called.

- *method execution/constructor execution:*
  An individual method or constructor is invoked.

- *field get:*
  A field of an object, class or interface is read.

- *field set:*
  A field of an object or class is set.

- *exception handler execution:*
  An exception handler is invoked.

- *class initialization:*
  Static initialization for class.

***Point Cut*** is "a set of *join point*s, plus, optionally, some of the values in the execution context of those *join point*s"[Greg2001].

AspectJ has several designators for *point cut*s. These *point cut*s can be composed by operators, like: && (and), ||(or), ~(not). As described in [Greg2001], typical *point cut* designators in AspectJ are:

- *calls(signature)/ executions(signature):*

  Used in situation when a method is called or a constructor is executed.

  The syntax of a method signature is:

  ResultTypeName RecvrTypeName.meth_id(ParamTypeName, ...)

  The syntax of a constructor signature is:

NewObjectTypeName.new(ParamTypeName, ...)

- *gets(signature)/gets(signature)[val]/sets(signature)/sets(signature)[oldVal]/sets(signature)[oldVal][newVal]*

  Uses for getting field's attributes (such as get variable's value) or setting field's attributes.

  The syntax of a field signature is:

  FieldTypeName ObjectTypeName.field_id

- *handles(ThrowableTypeName)*

  Uses for exception handler *join point.*

- *instanceof(CurrentlyExecutingObjectTypeName)/within(ClassName)/withincode(signature)*

  Uses for refer to an object/code is within Class Name or code is within the member of a method or constructor.

- *staticinitializations(TypeName)/initializations(TypeName)*

  Uses for class or object initialization.

***Advice*** is "a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut" [Greg2001]. *AspectJ* supports "before", "after", and "around" *advice*, each of which alters the source code in a different way.

Suppose a person wants to print out a message every time before one *join point*, he should use "before" *advice*. If he wants to do something after one *join point*, "after" *advice* should be used. If a person wants to run something at the same time as one *join point*, "around" *advice* is used.

***Weaving***

In AspectJ, *aspect weaving* combines the *aspect* component with Java program through the compiling process, so the Java source code is not impacted. Therefore, *weaving* in AspectJ provides a solution to add *aspect*s (monitoring points) in Java program, while at the same time the solution has no impact (non-intrusive) on the Java source code [Greg2001].

AspectJ carries out *aspect weaving* during its compiling process. The process consists of three steps: each *advice* is compiled into a standard method, the *join point*s part in the Java program is modified to fit the insertion of *advice*, and each *advice* method is inserted at the appropriate *join point*s in the Java program. After compiling, both *aspect* component and Java program are merged into one standard Java byte code file [Greg2001].

### 2.5.2 Crosscutting Example and Corresponding AspectJ Solution

Given the following *crosscutting concern* example:

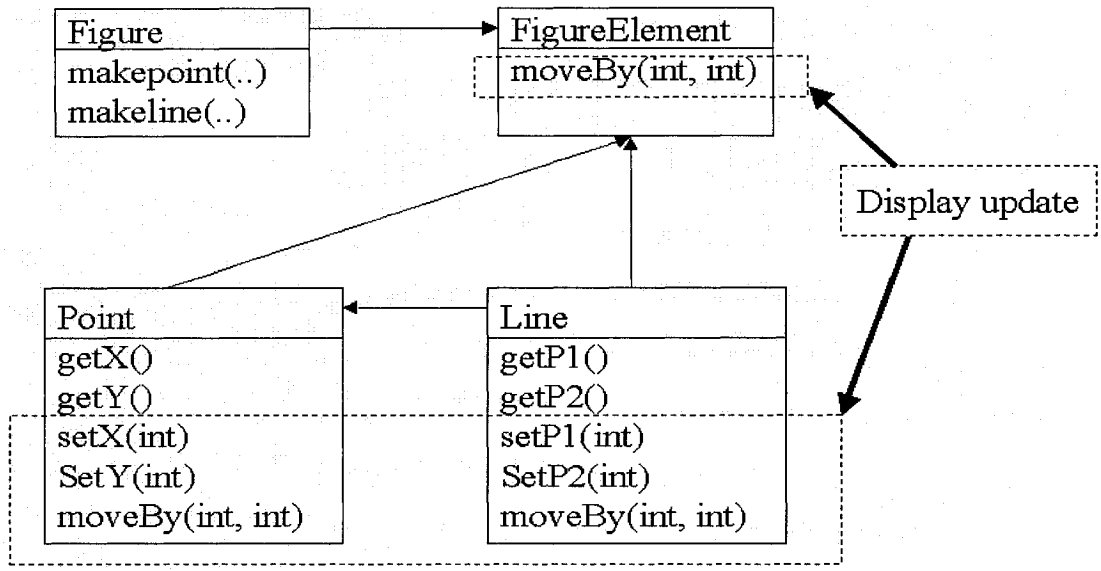



Figure 2.5.1 Crosscutting concern in Figure Editor[Greg2001]

In Figure 2.5.1, the display will be updated whenever there are activities to modify the data related to display. Display function is related to class "Figure", "FigureElement", "Point", and "Line". Operations that move elements in those classes are methods "moveBy", "SetX", "SetY", "SetP1", and "SetP2", which are implemented in three classes "FigureElement", "Point", and "Line". Since the display update function is distributed among different classes and methods, it leads to "tangling of code" and is considered as a *crosscutting concern* [Kicz1997].

In AspectJ, *crosscutting concern* related methods are regarded as components in *join point*s. An AspectJ program that provides an AOP solution to display *crosscutting* is listed in below:

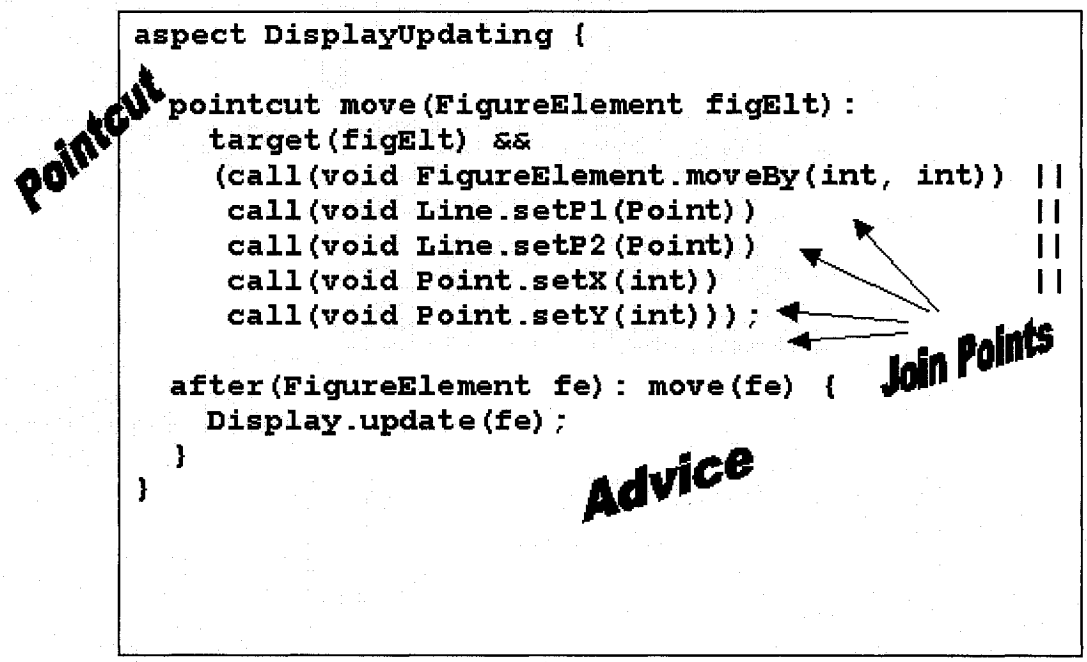



**Figure 2.5.2** AspectJ Solution[Greg2001]

In the example of Figure 2.5.2, five methods, moveBy, setP1, setP2, setX, setY, are regarded as *join points,* "move()" is defined as *point cut* designator for these *join points*. "display.update()" is an action whenever these *join points* are activated, and "after" *advice* is defined for this action so that it allows the action to be executed after these *join point*s are activated. As a result, when any method of SetX, SetY, moveBy, setP1, and setP2, is activated during the program execution, the display will be updated.

### 2.5.3 An AspectJ Sample Program

The AspectJ source code example in Figure 2.5.3 is a modified version of a program originally presented in [Ram2002]. The AspectJ program - MannersAspect will monitor the display method: *public static void say (String message),* and display some information before and after the display of "Hello, World!".

```
// HelloWorld.java
public class HelloWorld {
        public static void main(String[ ] args){
                say("Hello,World!");
        }
        public static void say(String message) {
                System.out.println(message);
        }
}

// MannersAspect.java
public aspect MannersAspect {
    pointcut callSay():call(* HelloWorld.say(..));
        before(): callSay() {
            System.out.println("Good Day");
        }
        after(): callSay() {
            System.out.println("Thank you");
        }
}
```

Join Point

Point Cut

Advice

**Figure 2.5.3** AspectJ HelloWorld sample program[Ram2002]

In Figure 2.5.3 example, callSay is defined as a *point cut*, which refers to *join point* HelloWorld.say:

```
pointcut callSay():call(* HelloWorld.say(..));
```

The behavior of the *PointCut* is defined as below

```
before(): callSay()

after(): callSay()
```

The behavior of the *advice* is to display the corresponding information "Good Day", "Thank you" before and after the execution of HelloWorld.Say method.

The weaved, compiled and executed *HelloWorld* AspectJ program will create the following output (Figure 2.5.4)

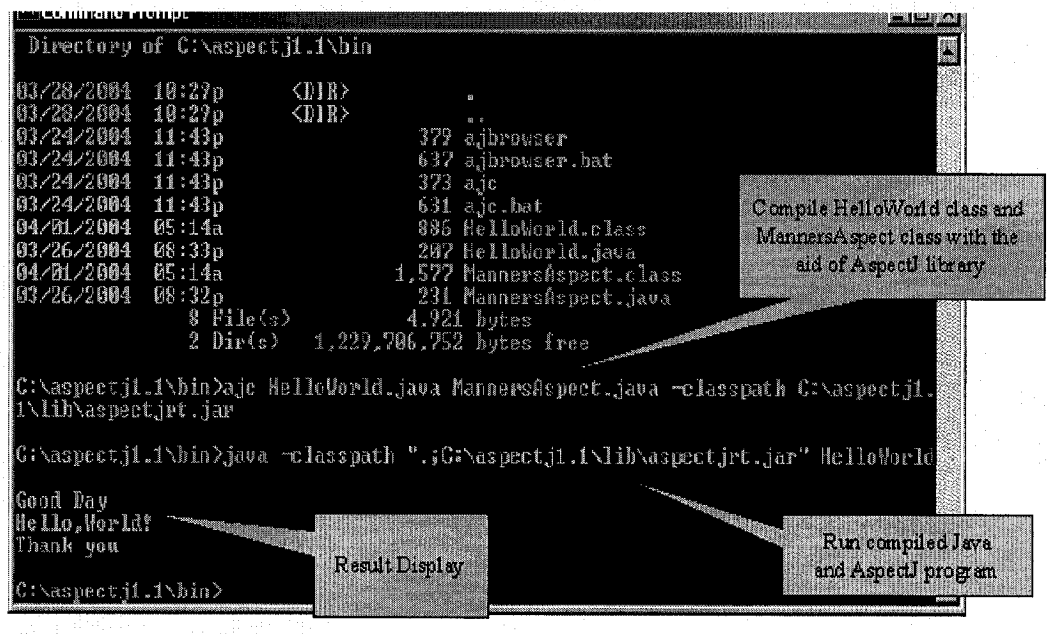



**Figure 2.5.4** View of HelloWorld compile and run

The program displays "Good day" before "Hello, World!", and displays "thank you" after "Hello, World!". The example illustrates how the *Aspect* implements and executes the added message during the program execution.

### 2.5.4 Applications of AspectJ

*Aspect* oriented programming enables a new approach to modularize all concerns of interest in a cleaner way compared with the more traditional object-oriented paradigm. As an *Aspect* implementation of Java, AspectJ enables better

encapsulation of distinct procedures and promotes future interoperation for Java program. The current design and implementation of AspectJ is upward compatible for Java, including platform compatible, tool compatible and programmer compatible.

### 2.5.4.1 Program Tracing in AspectJ

Program tracing is a typical application of AspectJ [Palo2002]. In this application, a tracing concern is regarded as a *crosscutting*, and an *aspect* is used to match and respond to the tracing concern. An *aspect* is implemented within AspectJ as a plug-in unit, which can be plugged in or removed from the program without any impact on the program itself. Therefore, the support of tracing in AspectJ has no direct impact (non-intrusive) on the original Java program.

As described earlier in section 2.3, source code instrumentation allows for monitoring a program execution by identifying entry and exit points of interest. Within AspectJ, these break points are captured by *aspects,* and the entry-exit routine is implemented in the *advice* part of the *aspect.* Therefore, the source code instrumentation tracing process is facilitated with the aid of AspectJ.

The example in Figure 2.5.5 is used to illustrate tracing in traditional Java applications. Suppose a trace class is created to output the trace message into the designated output stream. The member functions of this class are in below:

```
public class Trace {
    public static int TRACELEVEL = 0;
    public static void initStream(PrintStream s) {...}
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
}
```

**Figure 2.5.5** Member function of class Trace [Palo2002]

In traditional source code instrumentation tracing, one has to use traceEntry() and traceExit() methods everywhere in the existing Java code in order to trace the program and see the debugging messages. However, by using AspectJ, one can plug-in an *aspect,* which has the function of trace class, into the existing code to achieve the exactly same tracing functionality [Palo2002]. The sample code is in below:

```
Aspect TraceMyClasses {
        pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
// used within primitive designator to indicate that we only concerned the points where all
//the codes are inside of TwoDShape, Circle and Square classes

        pointcut myMethod(): myClass() && execution(* *(..));
// includes all the points at class's member functions

before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
        }
//Before class constructors, call traceEntry method, and display result along with JoinPoint's Signature message

after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
       }
// After class constructors, call traceExit method, and display result as well as JoinPoint's Signature message

before (): myMethod() {
Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
        }
// Before all the points at class member functions, call traceEntry method, and display result with JoinPoint's Signature message

after(): myMethod() {
Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
        }

// Before all the points at class member functions, call traceExit method, and display result along with JoinPoint's Signature message

  }
```

**Figure 2.5.6** AspectJ program with tracing function[Palo2002]

Figure 2.5.6 shows the corresponding AspectJ program with tracing function. In this example, there are three *point cut* designators and four *advice*s. All these *point cut* designators are user defined which represent a set of interested *join point*s.

The logical operators "||" and "&&" have the same semantic as in Java. The operators "." and "*" are wildcards supported by AspectJ to allow a more general definition of *pointcut*. The four *advices* in the program specify content, and location of the output streams. More specifically, "before" means the program takes an action before the method executes, and "after" means the program takes action after the method executes.

Whenever the program execution reaches these *join point*s, traceEntry() and traceExit() methods will be called and a message will be sent to the designated output.

### 2.5.4.2 Logging

Logging is another widely used application of AspectJ [Nich2002]. In traditional approaches, Log calls are manually inserted into every method; thus when the method is executed, Log calls are automatically triggered. Figure 2.5.7 shows a traditional Java example for implementing logging.

```
Logger.entry("doGet(...)");
JspTestController controller = new JspTestController();
controller.handleRequest(theObjects);
logger.exit("doGet");
```

**Figure 2.5.7** Traditional logging code[Nich2002]

In this implementation, logging statements are added to the source code by inserting these lines into all required methods. Developers have also to ensure that all method parameters are logged. Without exhaustive code review, these conventions are difficult to enforce.

Figure 2.5.8 illustrates how a similar logging functionality can be implemented using AspectJ (the parameter and return value logging are omitted to simplify the example).

```
Public aspect AutoLog{
 Pointcut publicMethods() : execution(public * org.apache.cactus..*(..));
// pointcut responds when any public method runs, since all method include "org.apache.cactus"

 pointcut logObjectCalls() :  execution(* Logger.*(..));
// includes execution of method contain all Logger

 pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
// include execution when any method which has a object call happens

 before() : loggableCalls(){
  Logger.entry(thisJoinPoint.getSignature().toString());
     }
// Before loggableCalls pointcut, execute Logger.entry
 after() : loggableCalls(){
  Logger.exit(thisJoinPoint.getSignature().toString());
     }
// After loggableCalls pointcut, execute Logger.exit

  }
```

**Figure 2.5.8** AspectJ Logging sample program[Nich2002]

In Figure 2.5.8, the string passed to the logger is derived from *thisJoinPoint*. *ThisJoinPoint* is a special reflective object in AspectJ, and it allows access to the run-time context in which the *join point* executes. In the actual *aspect*, the *advice* uses this object to retrieve the method parameters passed into each logged method call. Therefore, the logging *aspect* triggers log action, instead of adding logging procedure all the time [Nich2002].

### 2.5.4.3 Error and Exception Handling

AspectJ is an efficient programming style to address the error and exception handling problems in Java program [Bela2004]. Exception handling represents the points where errors were detected (in exception handlers and in method returns). In this situation, AspectJ provides a "throw" *join point* to capture the *join point* in case of exceptions [Bela2004].

It is important to evaluate the effect of exceptions in a system, but it is quite difficult to simulate certain exceptions in development situation. The reason for this is that the nature of the exception or the development environment is not an exact copy of the production environment [Bela2004]. For example, an exception occurring in the production environment due to database corruption cannot be simulated without knowing the exact corrupt command lines. Moreover, it may be impossible to capture a snapshot of the production database to transfer to the development server.

Besides simulating the exception to find out where the problem lies in the program, people also want to test if the code is strong enough to ensure that it can handle the problem correctly. This may lead to problems unless the exception is created in the patched code and its handling is observed. AspectJ allows for a simulation of throwing an exception as a result of calling a method on an object, without having to reproduce the exact runtime environments. A Java example of error handling is shown in below:

```
Package com.infosys.abhi;
Import java.io.*;
Public class ClassA
{
 public void methodA() throws IOException
 {
        System.out.println("Hello, World!");
 }
}

package com.infosys.bela;
public class ClassB
{
 public static void main(String[] args)
 {
        try
        {
                com.infosys.abhi.ClassA a = new com.infosys.abhi.ClassA();
                a.methodA();
                com.infosys.abhi.ClassC c = new com.infosys.abhi.ClassC();
                System.out.println(c.methodC());
        }
        catch (java.io.IOException e)
        {
                e.printStackTrace();
        }
        catch (Exception e)
        {
                e.printStackTrace();
        }
 }
}

package com.infosys.abhi;
public class ClassC
{
 public String methodC()
 {
        System.out.println("Hello, World!");
        return "Hi, World!";
 }
}
```

**Figure 2.5.9** Error handling sample Java program[Bela2004]

In the example shown in Figure 2.5.9, the method call methodA() in the class ClassA may cause a java.io.IOException. The goal is to identify an occurrence of exception of the methodA(), and it would not spend time and resources to set up specific environment to create an actual exception.

To resolve this problem, an *aspect* genExceptionA is introduced to trace the execution of methodA() and cause a java.io.IOException at runtime[Bela2004]. The *aspect* is used to test whether the application (ClassB) handles the exception or not. In addition, an "after" *advice* is used for point cut genExceptionAfterA(), so *point cut* genExeceptionAfterA() will execute following the execution of methodA().

In a real scenario, one may run an application to throw an unchecked exception, such as a subclass of java.lang.RuntimeException like NullPointerException, or an error such as a subclass of java.lang.Error like OutOfMemoryError. Both exception types are difficult to simulate in a development situation. But, the situation is able to be simulated by the *pointcut* genExceptionC and the corresponding "after" *advice*, since they cause the running code to cast out an OutOfMemory error following System.out.println().

The corresponding error handling AspectJ program is shown in Figure 2.5.10:

```
Package com.infosys.setl.aspects;
Public aspect GenException
{
 pointcut genExceptionA():
 execution(public void com.infosys.abhi.ClassA.methodA() throws java.io.IOException);
 pointcut genExceptionC():
 call(void java.io.PrintStream.println(String)) &&
 withincode(public String com.infosys.abhi.ClassC.methodC());
 pointcut genExceptionAfterA():
 call(public void com.infosys.abhi.ClassA.methodA() throws java.io.IOException);

 void around() throws java.io.IOException : genExceptionA()
 {
        java.io.IOException e = new java.io.IOException();
        throw e;
 }

 after() throws java.io.IOException : genExceptionAfterA()
 {
        java.io.IOException e = new java.io.IOException();
        throw e;
 }

 after() throws java.lang.OutOfMemoryError : genExceptionC()
 {
        java.lang.OutOfMemoryError e = new java.lang.OutOfMemoryError();
        throw e;
 }
}
```

**Figure 2.5.10** Error handling AspectJ program[Bela2004]

### 2.5.4.4 Security

AspectJ allows class hierarchy to be changed by adding new members to the class. This feature can be used to retrofit existing application with additional security features [Pawel2003].

**Authentication**

Authenticating the client application includes the following processes: configuring a login module, creating an instance of LoginContext, logging in, and logging out.

A Java program that implements authentication process is shown in Figure 2.5.11.

```
class BankClient {
        LoginContext lc = null;
        public static void main(String[] args) {
// Callback to get username and password. Required by LoginContext
        AppCallbackHandler handler = new AppCallbackHandler("scott", "echoman");
        try {
                lc = new LoginContext("Bank", handler);
                lc.login();
        } catch(LoginException e) {
// ...
                }
// ...
        BankHome homeBank = (BankHome) ctx.lookup("ejb/Bank");
        Bank bank = homeBank.create();
        System.out.println(bank.getAccountInfo("bill"));
// ...
        try {
                lc.logout();
        } catch(LoginException e) {
// ...
                }
        }
}
```

**Figure 2.5.11** Sample authentication Java program[Pawel2003]

The authentication process can be facilitated in AspectJ by encapsulating it in an *aspect*. An AspectJ solution for such an authentication process is shown in Figure 2.5.12.

```
class BankClient {
        public static void main(String[] args) {
// ...
        BankHome homeBank = (BankHome) ctx.lookup( "ejb/Bank " );
        Bank bank = homeBank.create();
        System.out.println( bank.getAccountInfo("bill") );
// ...
        }
}
aspect BankAspect {
        LoginContext lc = null;
        pointcut mainExecution():
        execution( public static void main( .. ) );
// Login before execution of main()
        before(): mainExecution() {
                AppCallbackHandler handler = new AppCallbackHandler( "scott", "echoman" );
        try {
                lc = new LoginContext( "Bank", handler );
                lc.login();
        } catch( LoginException e ) {
// ...
        }
}
// Logout after execution of main()
        after() returning: mainExecution() {
                try {
                        lc.logout();
                } catch( LoginException e ) {
// ...
            }
        }
}
```

**Figure 2.5.12** AspectJ authentication program [Pawel2003]

The *aspect* BankAspect is defined as the authentication (Log) process. BankAspect is triggered by the main method in client program. The before *advice* of BankAspect contains authentication processes configuring, creating client instance, and logging in. The after *advice* of BankAspect contains logging out process. Therefore, before the main method is activated, the authentication (login) process will be triggered and executed; after the execution of the main method, the logging out process will be executed.

### 2.5.5 Why Use AspectJ Tracing Facility for Source Code Instrumentation

The major benefits of AspectJ tracing facility for source code instrumentation are summarized below:

- *Aspects* work as "black box" for program functions by grouping program artifacts together and therefore it makes programs easy to comprehend, extend and reuse.
- *Aspect weaving* takes place during compiling process, so it makes the source code instrumentation method non-intrusive for original Java source code.
- Reduces *crosscutting* by introducing the notion of *aspect* [Kicz1997], and implement the inserting break points of source code instrumentation in *aspects* rather than modifying the original source code.
- The current design and implementation of AspectJ is compatible with different Java versions, operating system, platforms, and supporting development tools, without requiring original Java program to be modified.
- Source code instrumentation improves performance over run-time analysis, due to the reduction overhead (no additional run-time interpretation and therefore reduction of the virtual machine required).
- AspectJ provides a more powerful tracing functionality than Java so that it facilitates source code instrumentation.

# 3. Contributions

## 3.1 Motivation

Software maintenance is an important part of the software lifecycle with software comprehension playing a crucial role in performing software maintenance. Within the software comprehension community, three comprehension models are typically used to help people in creating a mental model of the program to be comprehended. These three models are top-down model, bottom-up model, and their combination-integrated model. In this research, we focus on the bottom-up comprehension model [Shnd1979], which is based on the assumption that the programmer fully understands all details of the program. Thus, analysis techniques are needed to collect and analyze the detail program information. Examples for such techniques are feature analysis [Kosc2000] and program slicing [Wei1982].

Feature analysis is applied to identify functional requirements as *features* in a system [Kosc2000]. *Slicing based feature analysis* [Sus2005] takes advantage of program slicing technique to analyze and identify dynamic features (see section 2.2.4).

This research aims to provide tracing support for features at the source code level. The statements included in these features were extracted through *slicing based feature analysis*. As one important tracing support technique [Ham2004], program instrumentation is employed in this research to capture the execution of features.

AspectJ is used as the source code instrumentation of choice because AspectJ facilitates insertion of break points in the instrumentation and provides a non-intrusive method for the instrumentation of Java source code (section 2.5.5). In addition, AspectJ is compatible with Java, which is the programming language applied in *slicing based feature analysis.*

### 3.1.1 Research Hypothesis

This research is motivated by the need to improve program comprehension. The goal of this research is to introduce a novel approach that utilizes feature analysis techniques to extract functional features from existing source code and to provide tracing support that captures the executions of these extracted features. This research takes advantage of the tracing support within AspectJ, and it leads to the following research hypothesis:

**Hypothesis**

Feature analysis techniques can be combined with AspectJ's tracing facility to capture feature executions and therefore improving program comprehension.

The hypothesis will hold since:

- Feature analysis is already a widely used technique in reverse engineering and program comprehension to recover functional features from the source code (section 2.2).

- It has been shown that program traces are an important factor to support certain program comprehension tasks [Rob2000].
- Previous work has illustrated that AspectJ has specific language support for source code instrumentation (section 2.5.5).

By combining the advantages of feature analysis, program instrumentation, and AspectJ tracing facility, it is possible to capture feature information within traces and to improve the comprehension of features and their executions.

## 3.2 Detailed Research Contribution

Based on this research hypothesis, a source code instrumentation method based on AspectJ's tracing facility is introduced. The contributions of this research can be summarized as the following:

- Reviewed existing program instrumentation approaches (section 2.3).
- Present a source code instrumentation approach that traces feature analysis results based on AspectJ's tracing capability.
- Identify mapping rules from Java to AspectJ to support the tracing of features.
- Identify exception cases where no mapping rules can be created.
- Complete a software tool - AspectJTracer to semi-automatically create AspectJ tracing program, which traces Java program by mapping each feature of interest to an *aspect*.
- Perform initial case study to validate the presented approach.

## 3.3 AspectJTracer Tool

### 3.3.1 AspectJTracer System Design

The AspectJTracer tool is implemented within the CONCEPT project [Ril2002]. AspectJTracer utilizes some analysis techniques provided by the underlying CONCEPT project. An overall architecture of the source code instrumentation approach and its integration within the CONCEPT project is completed and illustrated in Figure 3.3.1.

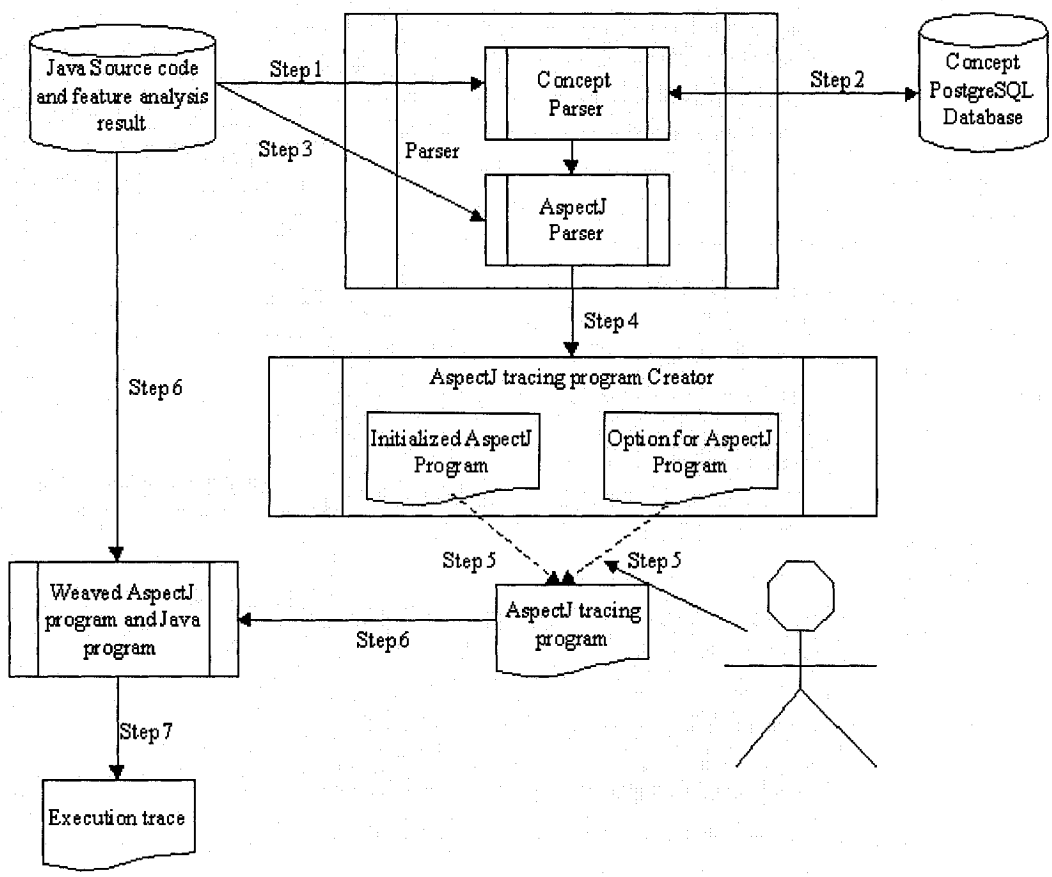



**Figure 3.3.1** Workflow of AspectJTracer

The presented approach is built on top of two parsers and *AspectJ tracing program Creator*. The parsers include an *AspectJ parser* and a *CONCEPT parser*. The *CONCEPT parser*, an existing implementation within the CONCEPT project, reads Java source code and stores abstract syntax information in the *CONCEPT PostgreSQL database*. The *AspectJ parser* reads the Java source code, the result of feature analysis, and supplementary information from *CONCEPT parser*. The *AspectJ parser* is developed as part of this project to provide *AspectJTracer* the information that required by the tracing program.

*AspectJ tracing program Creator* creates AspectJ *point cut* definitions based on feature related statements. *Initialized AspectJ program* and *Option for AspectJ program* are created to help user to complete the AspectJ tracing program.

The following steps are applied:

**Step 1.** *CONCEPT parser* reads the original Java source code.

**Step 2.** Abstract syntax tree generated by *CONCEPT parser* is stored in the *CONCEPT PostgreSQL database*.

**Step 3.** The *AspectJ parser* reads the Java source code and the statements generated by feature analysis.

**Step 4.** *AspectJ tracing program Creator* establishes a mapping from the feature related statements to an initialized AspectJ tracing program based on the syntax information from both parsers. The mapping rules are discussed in more details in section 3.3.4.

**Step 5.** In cases where user interaction is required to complete the AspectJ tracing program, the user can follow the available templates. This part will be discussed in section 3.3.3.

**Step 6.** The Java program and the AspectJ tracing program are weaved together for checking error and creating executable program.

**Step 7.** Executing the weaved program produces the execution trace. The execution trace information includes: statement content, statement location, parameter value, statement attributed class, statement attributed method, method parameter, etc.

### 3.3.2 CONCEPT Project

The research presented in this thesis is an integrated part of the CONCEPT project (Comprehension Of Net-CEntered Programs and Techniques) [Ril2002]. The major goal of the CONCEPT project is to address the current and future challenges related to the comprehension of large and distributed systems at the source code and architectural level. Its objective is to provide programmers with novel comprehension approaches based on different source code analysis, visualization, reverse engineering, and architectural recovery techniques as well as their applications. The CONCEPT Architecture is shown in Figure 3.3.2.

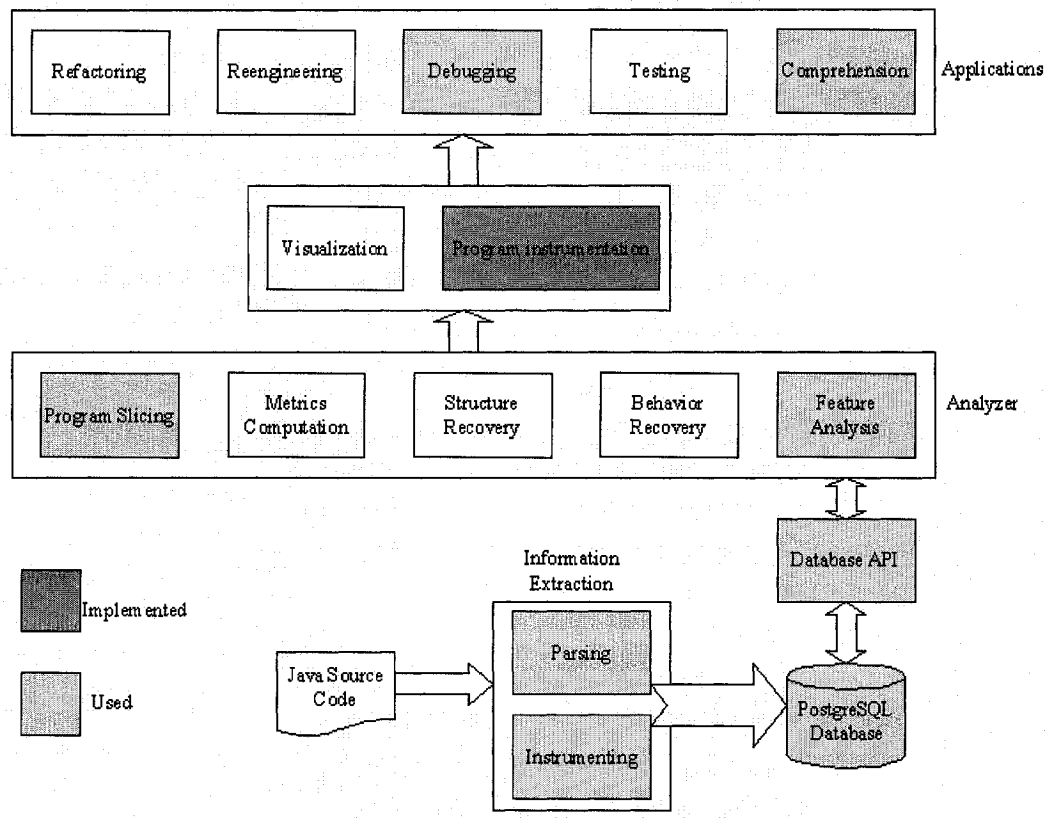



**Figure 3.3.2** Concept Architecture

The CONCEPT environment was implemented using layered system architecture, with the PostgreSQL database acting as a repository for all shared data.

The database stores information extracted by parsing (in the form of an abstract syntax tree) and instrumentation of Java source code (dynamic information). This information is accessed by the different components of the Analysis layer using the Database API.

In this research, parts of the CONCEPT project will be reused. The result of *slicing based feature analysis* [Sus2005] is the input of current research. In addition,

*CONCEPT Parser*, PostgreSQL database, and database API, are also used in current research.

### 3.3.3 AspectJTracer Design

The AspectJTracer tool has been designed and implemented to create the tracing program. AspectJTracer is based on *slicing based feature analysis* performed within the CONCEPT project, and it involves the following four major steps:

- *Input*: input the result of feature analysis (related statement number) and source code
- *Create Initialized tracing program*: Create initialized AspectJ tracing program on feature analysis result
- *Modify tracing program*: Modify the AspectJ tracing program by referring to *options for AspectJ program* and software tool's help
- *Compile*: Compile the AspectJ tracing program and Java program for error checking

The overall AspectJTracer workflow is depicted in Figure 3.3.3:

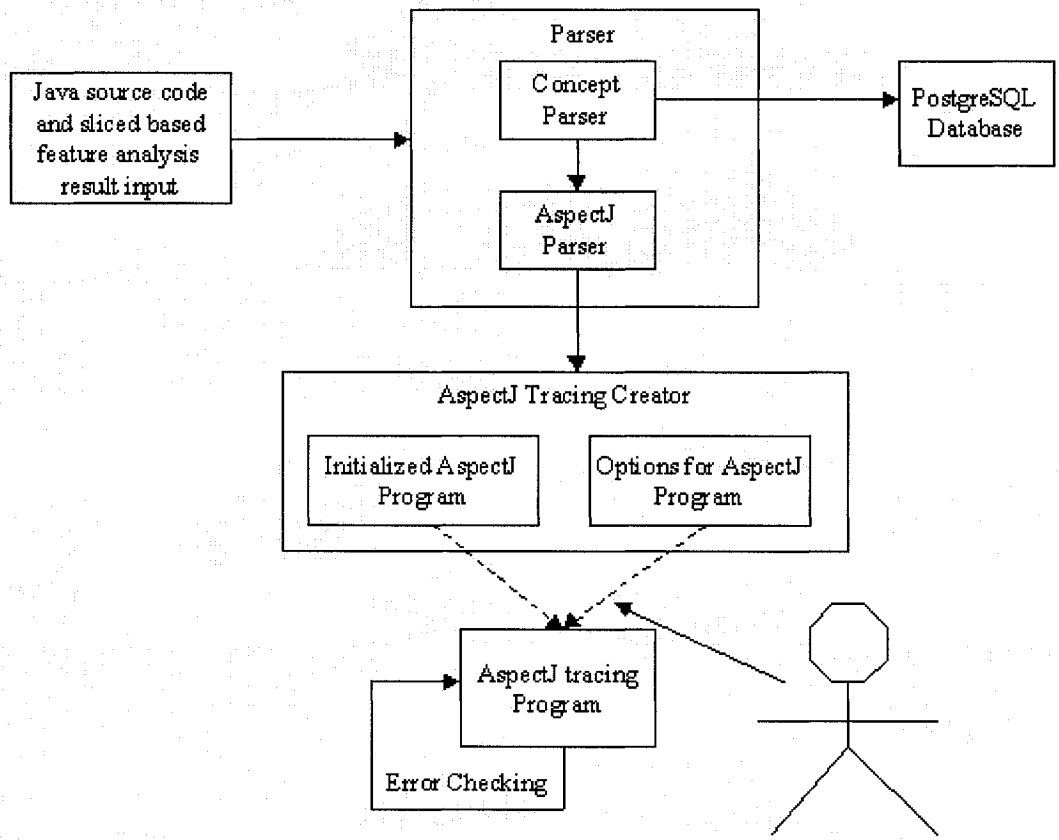



**Figure 3.3.3** AspectJTracer workflow

The *CONCEPT parser* is responsible for analyzing syntax information of Java source code, and provides the *AspectJ parser* with required syntax information.

The *AspectJ parser* reads both the Java source code and the feature analysis result to establish a semi automatic mapping from the source code to the tracing program.

The detailed AspectJTracer workflow is as following:

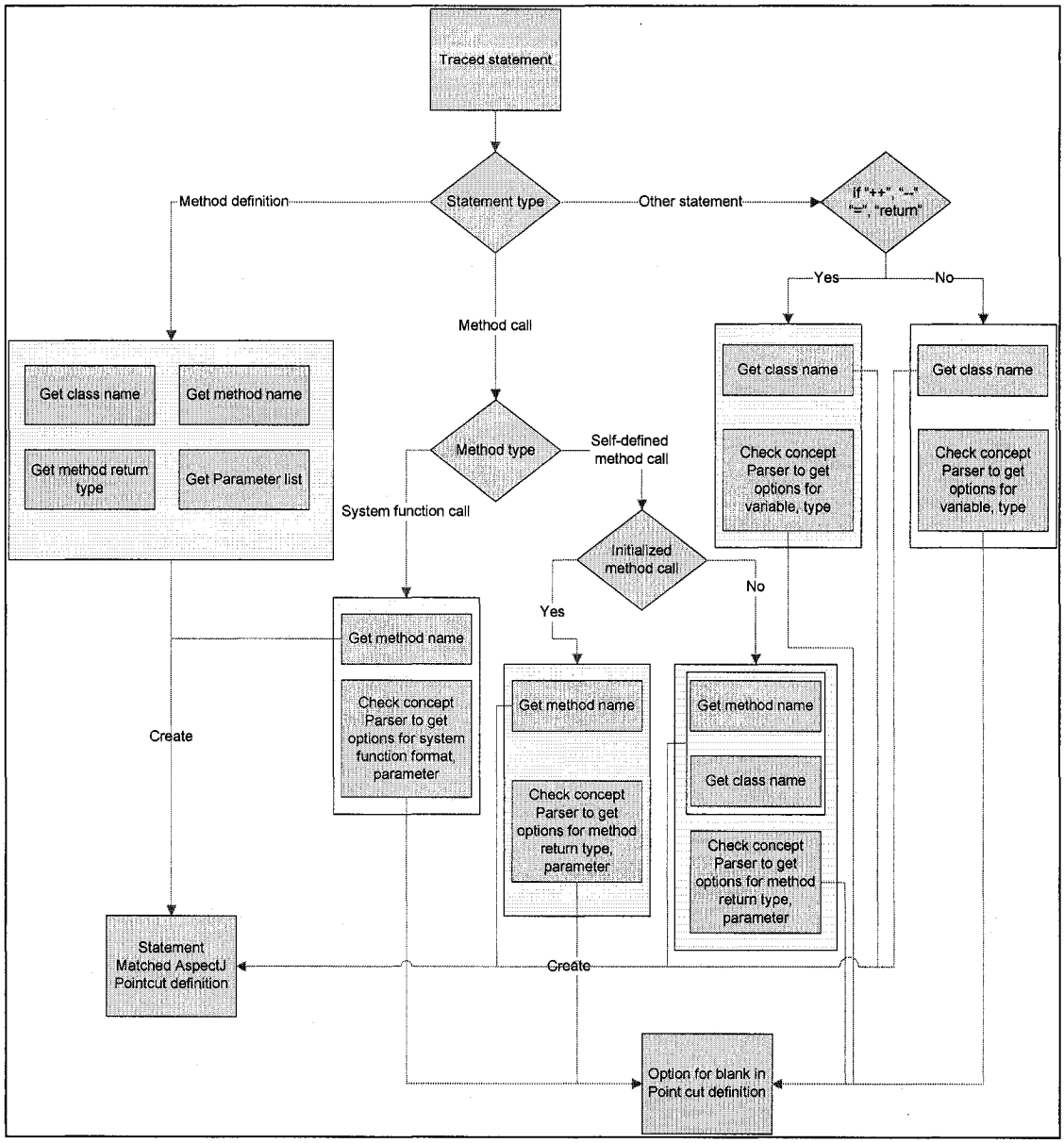



**Figure 3.3.4** AspectJTracer detail workflow for *point cut* definition

In Figure 3.3.4, AspectJTracer divides Java statement into three types, and constructs the corresponding *point cut* and *advice* part in the tracing *aspect*. Java statement types are classified into method declaration, method calls, and other statements. The detailed discussion is in section 3.3.4.

### 3.3.3.1 Basic Constructs of the AspectJTracer

For method declaration statements, AspectJTracer extracts method name, class name, method return type and parameter type list, and constructs the corresponding AspectJ *point cut* definition. This process is implemented automatically.

For a method call statement, it is first required to identify if it is a Java system function method call, an initialized method call, or a general method call. For the first two types, method name is extracted and AspectJ *point cut* definition is constructed; for general method call, both method name and class name are extracted to construct AspectJ *point cut* definition. *AspectJ parser* also extracts syntax information from *CONCEPT parser* and constructs corresponded optional items in *Options for AspectJ Program.*

For other statement type, if the statement includes variable setting, "set" *point cut* designator is used; "get" *point cut* designator is utilized for variable getting; "handler" *point cut* designator is used for exception handing. Then both method name and class name are extracted from context, and *point cut* definition is constructed. *AspectJ parser* extracts syntax information from *CONCEPT parser* and constructs corresponded optional items in *Options for AspectJ Program.*

The AspectJ tracing program format is as Figure 3.3.5:

```
Aspect Aspect_class{
        before(): pointcut1 {//map feature analysis based statement 1
        advice1 // get tracing information of statement 1, and write to trace file
        }
        before(): pointcut2 {//map feature analysis based statement 2
        advice2 // get tracing information of statement 2, and write to trace file
        }
        before(): pointcut3 {//map feature analysis based statement 3
        advice3 // get tracing information of statement 3, and write to trace file
        }
        ...
        write_to_tracefile();//write tracing information into trace file
        }
```

**Figure 3.3.5** Tracing program structure

*Point cut* definition needs to be changed according to different statement types, and the rule is discussed in section 3.3.5. *Advice* uses AspectJ *thisjoinpoint* function to refer to the traced statement content, such as statement location, argument, etc. *thisJoinPoint*, a special reference variable in AspectJ, contains reflective information about the current *join point* for the *advice* to use [Palo2002]. "Before" *point cut* is applied to allow the *advice* be executed before the *join point* reaches.

In *advice*, a counter is used when a statement belongs to a loop, and only the first execution of this statement is recorded into the trace file. The purpose of this mechanism is to prevent a loop statement from creating multiple entries in the trace file.

*Abstract aspect* is used in situation where *point cut* overriding and *advice* inheritance are required [Greg2001]. In this research, the *advice* part is created automatically by

AspectJTracer tool, so it does not require being inherited and extended. Therefore, *abstract aspect* is not applied in this research.

### 3.3.3.2 Multi-class & multi-feature Solution

For the presented approach, each feature-related statement is matched by one *point cut* definition. In the case that multiple features are traced, the approach requires each feature be captured in a separate *aspect* (Figure 3.3.6).

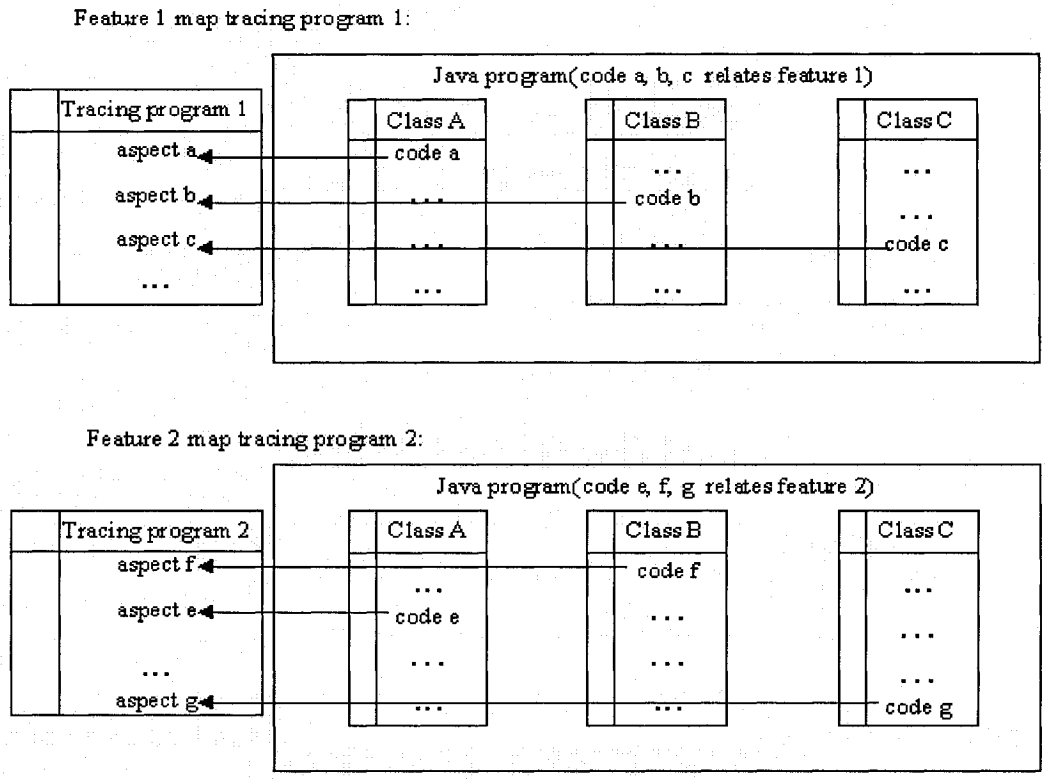



**Figure 3.3.6** Multiple tracing programs map multiple features across multiple classes

Figure 3.3.6 illustrates such a multi-features tracing. The Java program includes three classes: class A, class B, and class C. Two features are distributed across this program: feature 1, and feature 2.

Code a in class A, code b in class B, code c in class C are all related to feature 1. AspectJTracer creates the tracing program 1 for tracing feature 1. In the tracing program 1, *aspect* a matches code a, *aspect* b matches code b, *aspect* c matches code c. The sequence of *aspect* a, b, c is not considered in this approach as it has no impact on the tracing. Therefore, tracing program 1 has three parts: *aspect* a, *aspect* b, and *aspect* c.

Code e in class A, code f in class B, code g in class C are all related to feature 2. AspectJTracer creates the tracing program 2 that traces feature 2. In this case, *aspect* e matches code e, *aspect* f matches code f and *aspect* g matches code g. The tracing program 2 contains three parts: *aspect* f, *aspect* e, and *aspect* g.

### 3.3.4 Mapping Rules from Java Statements to AspectJ *Point Cuts*

As discussed before, each feature-related statement in the original program is mapped to one *point cut* in the tracing program. In order to establish the mapping from Java statements to AspectJ *point cuts*, Java statements are categorized according to their Java syntax [Java1996], and Java statement types for which a mapping is applicable are derived (Figure 3.3.7).

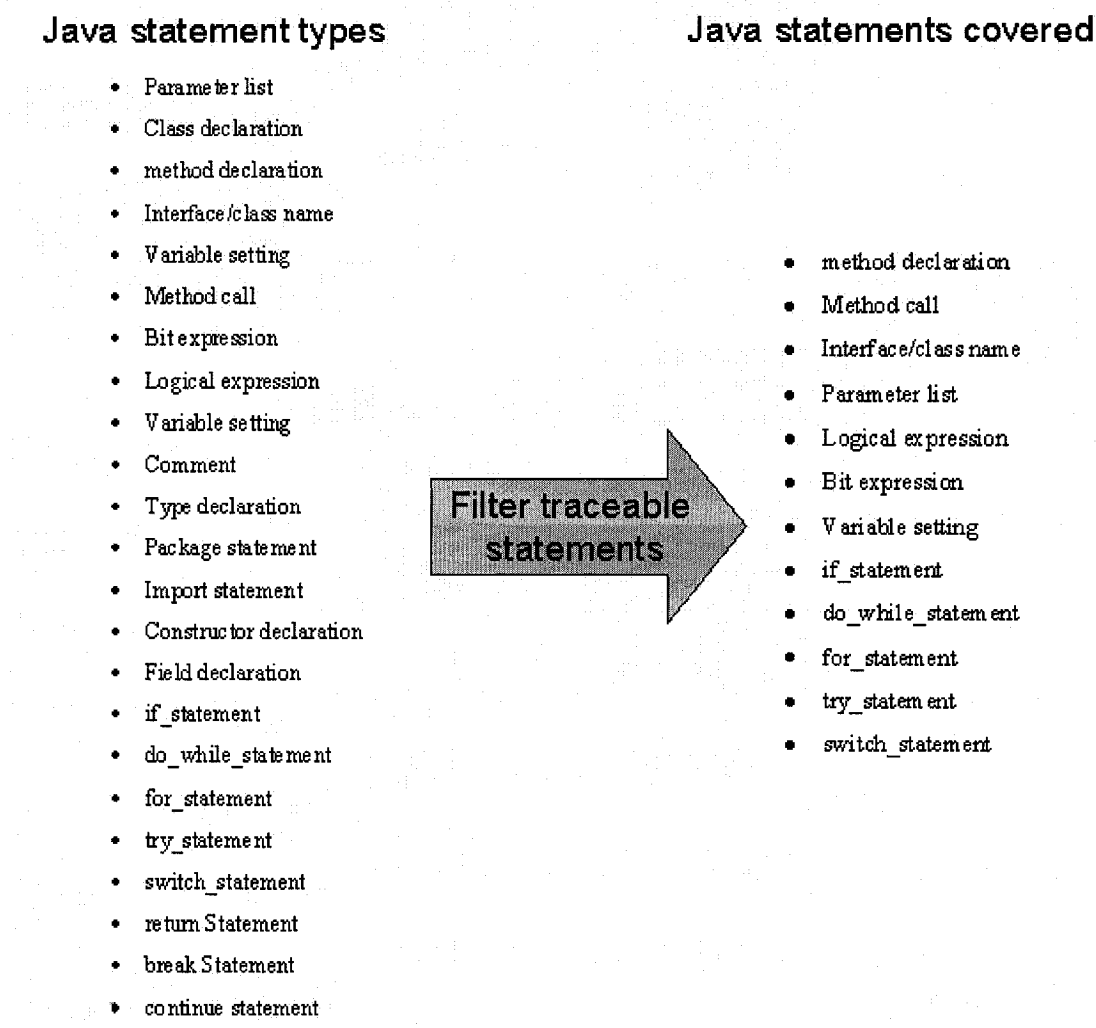



**Figure 3.3.7** AspectJ traceable Java statement types

The detailed mapping rule for each statement type is discussed in below.

- **Method declaration**

*Given:*

return_value_type method_name(parameter_type1 parameter1, parameter_type2 parameter2...)

*Mapping rule*:

Method name, method return type, parameters, and parameter type are extracted from method declaration statement. The corresponding class name needs to be extracted from context, and "execution" *point cut* designator is applied to monitor the execution of the method. Thus, the *point cut* format of method declaration is:

execution(return_value_type class_name.method_name(par_type1, par_type2...))

*Limitations/Restrictions:*

*None*

*Example:*

*Original:*

```
void deposit(double amount){
```

*AspectJ:*

```
execution(double account.deposit(double))
```

- **Method call**

*Given:*

method_call (parameterlist)

*Mapping rule:*

Method name and parameters are extracted from method call statement. The corresponding class name, method return type, and parameter type need to be extracted from context, and "call" *point cut* designator is applied to monitor method call. Thus, the *point cut* format of method call is:

call ( return_value_type class_name.method_name (parameter_type1,parameter_type2))

*Limitations/Restrictions:*

Not applicable for class initialization method call

*Example:*

*Original:*

```
double currentbalance=account.deposit(amount);
```

*AspectJ:*

```
call(double account.deposit(double))
```

- **Class initialization method call**

*Given:*

method_call (parameterlist)

*Mapping rule:*

Method name and parameters are extracted from current statement. In class initialization method call statement, the corresponding class name is the same as method call. Method return type and parameter type need to be extracted from context, and "call" *point cut* designator is applied to monitor the initialized method call. Thus, the *point cut* format of class initialization method call is:

call ( return_value_type class_name.new (parameter_type1, parameter_type2...))

*Example:*

*Original:*

```
account account = new account(100);
```

*AspectJ:*

```
call(public account.new(double))
```

- **Parameter list**

*Given:*

parameterlist

*Mapping rule:*

Parameter type needs to be extracted from context, and "args" *point cut* designator is applied since it declares parameter types. Thus, the *point cut* format of parameter list is:

args( parameter_name)

*Limitations/Restrictions:*

None

*Example:*

*Original:*

```
System.out.println("The current balance after withdraw/deposit is: ");
```

*AspectJ:*

```
String notice="The current balance after withdraw/deposit is:";
call(* java.io.PrintStream.println(String))&&args(notice)
```

- **Bit expression**

*Given:*

variable++|--

*Mapping rule:*

Variable name and "++" or "--" are extracted from bit expression statement. Variable type and the corresponding class name need to be extracted from context, and "set" *point cut* designator is applied to monitor variable setting. Thus, the *point cut* format of bit expression is:

set (variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

```
balance++;
```

*AspectJ:*

```
set (double account.balance)
```

- **Variable setting**

*Given:*

variable = expression;

*Mapping rule:*

Variable name is extracted from current statement. Variable type and the corresponding class name need to be extracted from context, and "set" *point cut* designator is applied to monitor variable setting. Thus, the *point cut* format of variable setting is:

set (variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

balance=accountbalance;

*AspectJ:*

set (double account.balance)

- **Logical expression**

*Given:*

Logical expression

*Mapping rule:*

Variable name is extracted from the Logical expression. Variable type and the corresponding class name need to be extracted from context, and "get" *point cut* designator is used to monitor variable getting. Thus, the *point cut* format of logical expression is:

get(variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

qty<30

*AspectJ:*

get (double account.qty)

- **If_statement**

*Given:*

if (expression)

*Mapping rule:*

Variable name needs to be extracted from expression, variable type and the corresponding class name need to be extracted from context, and "get" *point cut* designator is applied to monitor variable getting. Thus, the *point cut* format of if statement is:

get(variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

if (balance>accountbalance)

*AspectJ:*

get (double account.balance)

- **Do_while_statement**

*Given:*

while(expression)

*Mapping rule:*

Variable name needs to be extracted from expression, variable type and the corresponding class name need to be extracted from context, and "get" *point cut* designator is applied to monitor variable getting. Thus, the *point cut* format of Do_while_statement is:

get(variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

while (balance>accountbalance)

*AspectJ:*

get (double account.balance)

- **For_statement**

*Given:*

for (expression)

*Mapping rule:*

Variable name needs to be extracted from expression, variable type and corresponding class name need to be extracted from context, and "get" *point cut* designator is applied to monitor variable getting. Thus, the *point cut* format of for_statement is:

get(variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

```
for(qty=0;qty<30; qty++)
```

*AspectJ:*

```
get (double account.qty)
```

- **Try_catch_statement**

*Given:*

catch(exception)

*Mapping rule:*

This statement includes exception and catch. Exception type needs to be extracted from expression, and "handler" *point cut* designator is applied to monitor exception handling. Thus, the *point cut* format of try_catch_statement is:

handler(exception_type)

*Limitations/Restrictions:*

*none*

*Example:*

*Original:*

catch(IOException e)

*AspectJ:*

handler(IOException)

- **Switch_statement**

*Given:*

switch(expression)

*Mapping rule:*

Variable name needs to be extracted from expression, variable type and corresponding class name need to be extracted from context, and "get" *point cut* designator is applied to monitor variable getting. Thus, the *point cut* format of switch_statement is:

get(variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

switch(qty)

*AspectJ:*

get (int account.qty)

- **Return Statement**

*Given:*

return(expression)

*Mapping rule:*

Variable name needs to be extracted from expression, variable type and corresponding class name need to be extracted from context, and "get" *point cut* designator is applied to monitor variable getting. Thus, the *point cut* format of return_statement is:

get(variable_type class_name.variable_name)

*Limitations/Restrictions:*

The variable must be public member of class

*Example:*

*Original:*

return(qty)

*AspectJ:*

get (double account.qty)

Other statements, such as break statement, continue statement, package statement, are not traceable in AspectJ syntax.

Therefore, all Java statement types covered by the AspectJTracer rules fall in one of these three main mapping categories:

- Method declaration: method declaration
- Method call: method call , class initialization method call
- Other statement: parameter list, logical expression, Bit expression, Variable setting, If statement, Do while statement, For statement, Try statement, Switch statement, Return statement.

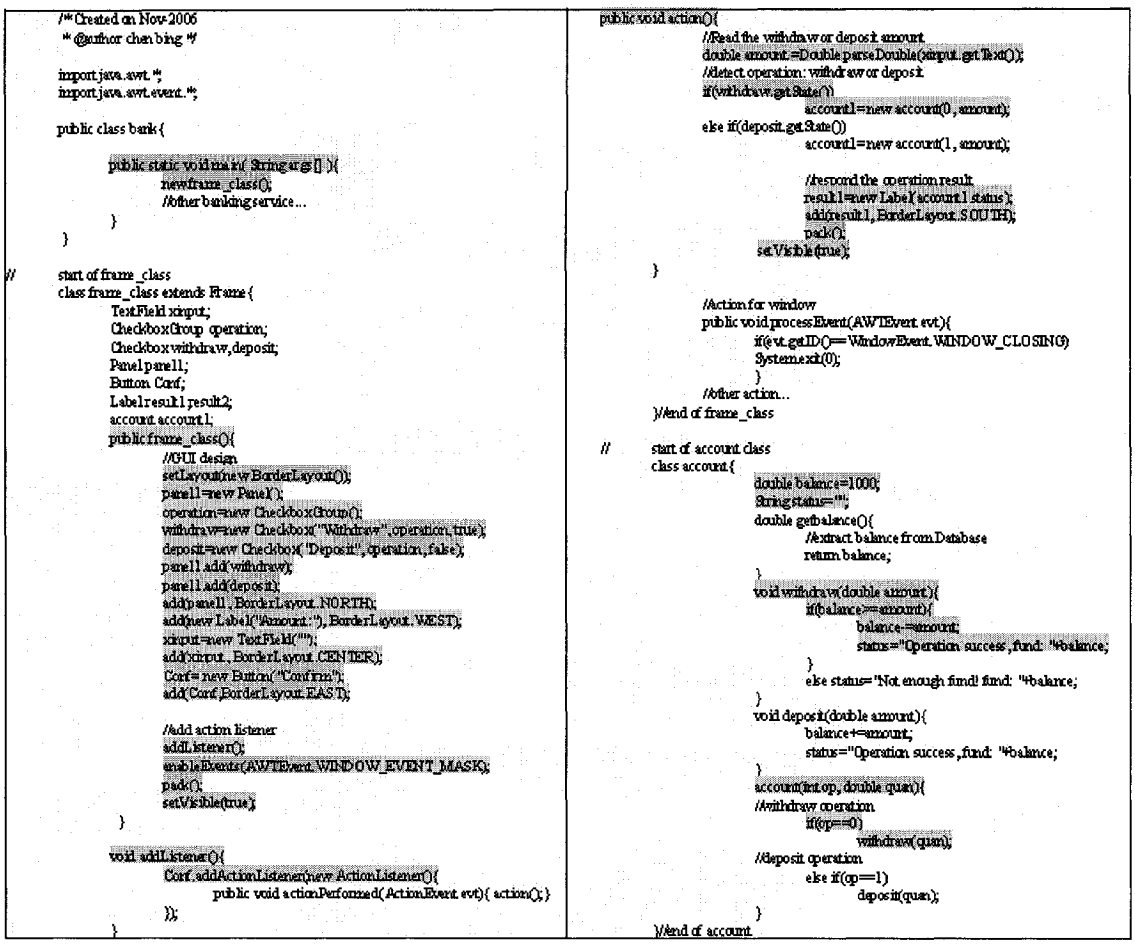
# 4. Case Study

A case study was completed to validate the source code instrumentation approach presented in chapter 3 of this thesis. Based on a sample Java program, this case study is used to illustrate the use of the AspectJTracer and the process of creating the tracing program. At the end of this chapter, benefits and limitations of the presented approach are discussed.

## *4.1 Java Source Code and Feature in Case Study*

The Java program used in this case study simulates ATM banking operation, allows user to select "withdraw" or "deposit" operation, then gives the corresponding operation result. Please note that the complete source code of this program can be found in Appendix A.

In this sample program, two functional features are extracted by applying *slicing based feature analysis*. The first feature extracted corresponds to "withdraw" operation and the second feature is "deposit" operation. The tracing program for the first feature is used in the following section to illustrate the use of the AspectJTracer and its application. Statements that relate to this feature are highlighted in Figure 4.1.1:



```
/* Created on Nov 2006
 * @author chen bing */

import java.awt.*;
import java.awt.event.*;

public class bank {

        public static void main( String args[] ){
                newframe_class();
                //other banking service...
        }
}

//      start of frame_class
class frame_class extends Frame {
        TextField xinput;
        CheckboxGroup operation;
        Checkbox withdraw,deposit;
        Panel panel1;
        Button Conf;
        Label result1,result2;
        account account1;
        public frame_class(){
                //GUI design
                setLayout(new BorderLayout());
                panel1=new Panel();
                operation=new CheckboxGroup();
                withdraw=new Checkbox("Withdraw",operation,true);
                deposit=new Checkbox("Deposit",operation,false);
                panel1.add(withdraw);
                panel1.add(deposit);
                add(panel1,BorderLayout.NORTH);
                add(new Label("Amount:"),BorderLayout.WEST);
                xinput=new TextField("");
                add(xinput,BorderLayout.CENTER);
                Conf=new Button("Confirm");
                add(Conf,BorderLayout.EAST);

                //add action listener
                addListener();
                enableEvents(AWTEvent.WINDOW_EVENT_MASK);
                pack();
                setVisible(true);
        }

        void addListener(){
                Conf.addActionListener(new ActionListener(){
                        public void actionPerformed(ActionEvent evt){ action(); }
                });
        }

public void action(){
                //Read the withdraw or deposit amount
                double amount=Double.parseDouble(xinput.getText());
                //detect operation: withdraw or deposit
                if(withdraw.getState())
                        account1=new account(0,amount);
                else if(deposit.getState())
                        account1=new account(1,amount);

                        //respond the operation result
                        result1=new Label(account1.status);
                        add(result1,BorderLayout.SOUTH);
                        pack();
                        setVisible(true);
        }

                //Action for window
                public void processEvent(AWTEvent evt){
                        if(evt.getID()==WindowEvent.WINDOW_CLOSING)
                        System.exit(0);
                        }
                //other action...
        }//end of frame_class

//      start of account class
        class account{
                        double balance=1000;
                        String status="";
                        double getbalance(){
                                //extract balance from Database
                                return balance;
                        }
                        void withdraw(double amount){
                                if(balance>=amount){
                                        balance-=amount;
                                        status="Operation success,fund: "+balance;
                                }
                                else status="Not enough fund! fund: "+balance;
                        }
                        void deposit(double amount){
                                balance+=amount;
                                status="Operation success,fund: "+balance;
                        }
                        account(int op, double quan){
                        //withdraw operation
                                if(op==0)
                                        withdraw(quan);
                        //deposit operation
                                else if(op==1)
                                        deposit(quan);
                        }
        }//end of account
```

**Figure 4.1.1** Feature related statements (highlighted) in source code

In order to trace the feature, it is necessary to establish a mapping of each identified feature statement and its corresponding tracing syntax in AspectJ. Each statement is considered as one *join point*, and the corresponding *point cut* definition is constructed based on the mapping rules presented in section 3.3.4. An *advice* is used to record the statement information, such as statement location, relevant class name, parameter, method name, etc.

Some sample *point cut* definitions are shown in the following figure:

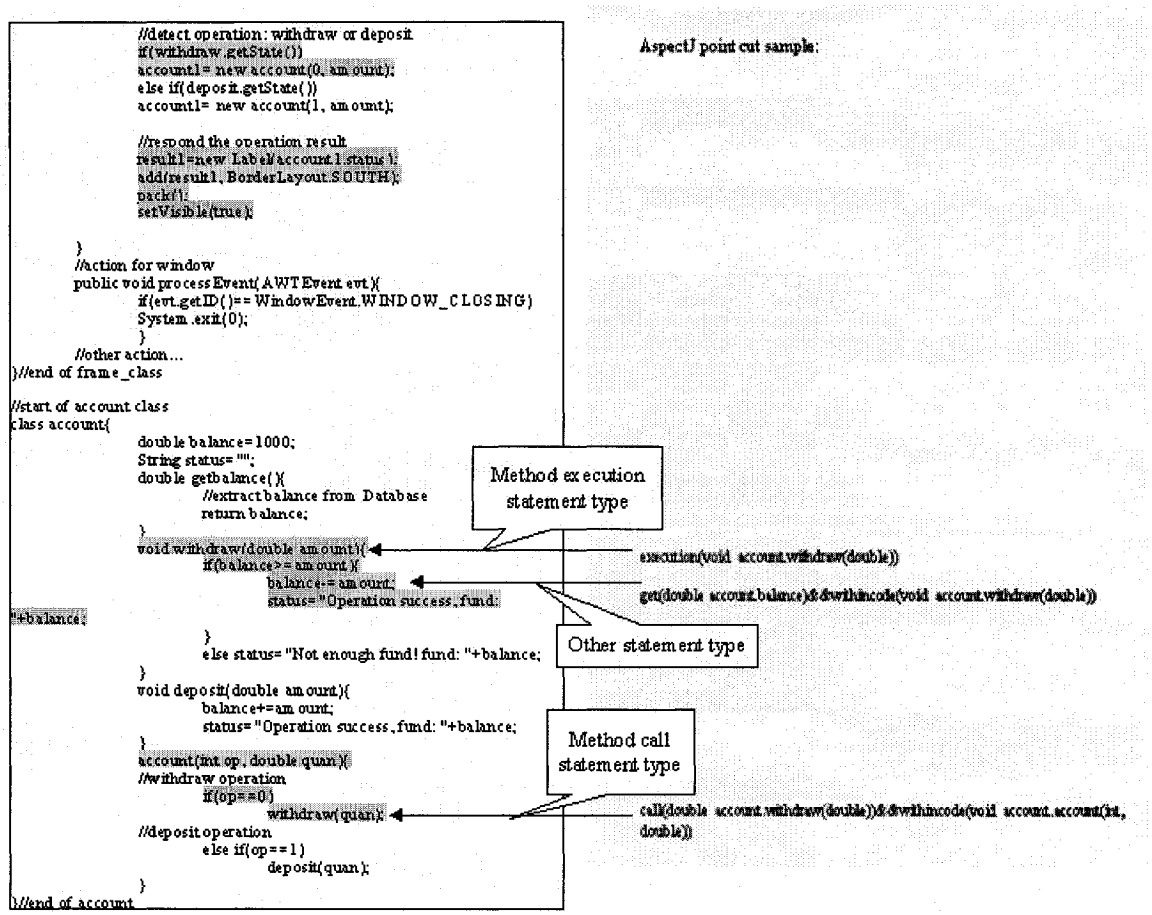



**Figure 4.1.2** AspectJ tracing program *point cut* definition sample

The complete tracing program for this feature can be found in Appendix B.

## *4.2 AspectJTracer Usage*

### 4.2.1 Environment and Development Tool

In this research, the operating system used is Windows 2000, and the development environment is *Eclipse* 3.0.1, which includes AspectJ API plug-in - AspectJ Development Tools (AJDT) 1.2.

### 4.2.2 AspectJTracer Menu

Figure 4.2.1 shows an overview of the system function provided within the AspectJTracer. The functions of this software tool can be summarized in four major categories: File, Edit, Tools (Creation and Edit, Syntax Option), and Help.

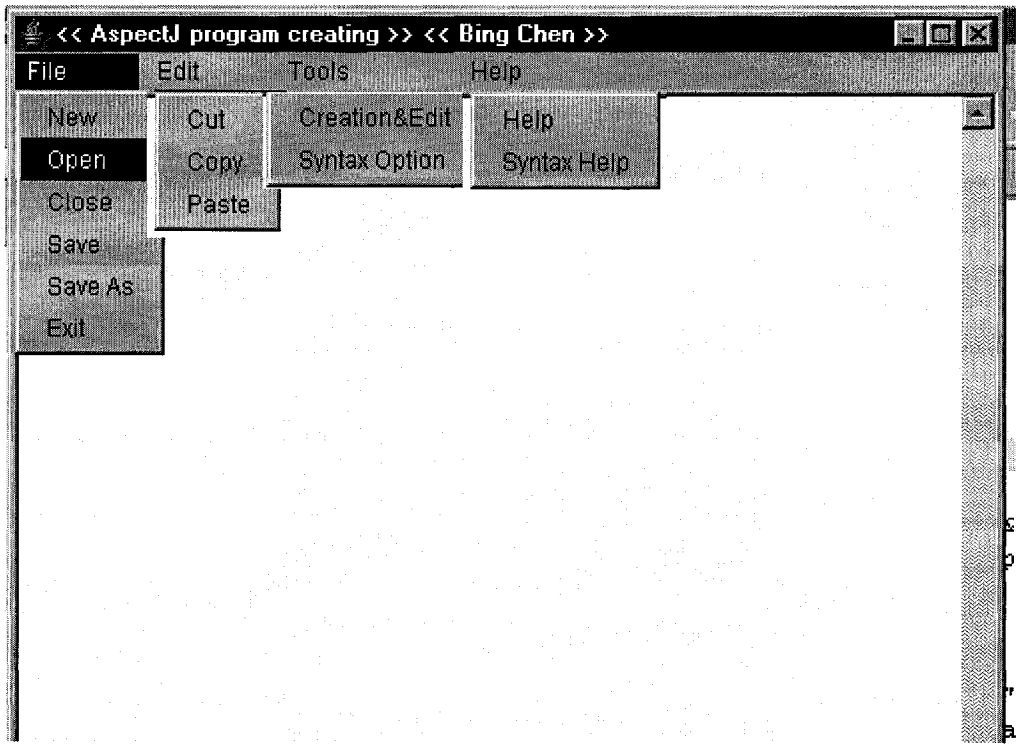



**Figure 4.2.1** AspectJTracer menu

Figure 4.2.1 shows an overview of the core functionality of the AspectJTracer tool. "File" concerns input of source code and feature. "Edit" is used to edit program. In "Tools" menu, "Creation&Edit" aims to create and edit initialized AspectJ tracing program; "Syntax Option" provides optional items for the AspectJ tracing program. "Help" provides software tool help; "Syntax Help" provides syntax mapping rule from Java statement to AspectJ *point cut* definition.

The process of creating a sample tracing program is depicted in section 4.2.3, 4.2.4.

### 4.2.3 Input of Source Code and Feature

After user clicks menu "File", "Open", and selects the Java program, an editing environment will display. See Figure 4.2.2.

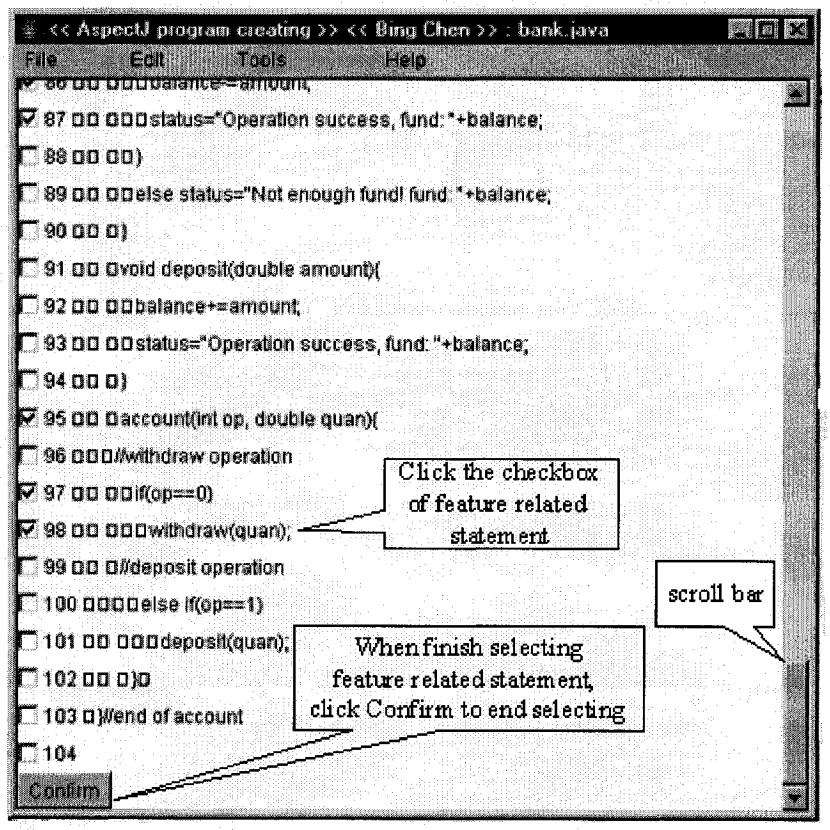



**Figure 4.2.2** Input View

In this view, user checks the statements that are the result of feature analysis. After finishing selecting, user needs to click button "Confirm" to finish the input process. In the case study example presented in section 4.1, statement numbers that are identified by feature analysis are: 9, 10, 24, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 41, 43, 44, 47, 52, 54, 56, 57, 62, 63, 64, 65, 78, 79, 84, 85, 86, 87, 95, 97, and 98.

### 4.2.4 Create and Edit AspectJ Tracing Program

Users select the menu "Tools", and "Creation&Edit", which will perform the mapping (based on the predefined mapping rules) of the features in the original Java source code to their trace capturing in AspectJ. The initialized AspectJ tracing program is created as following figure:

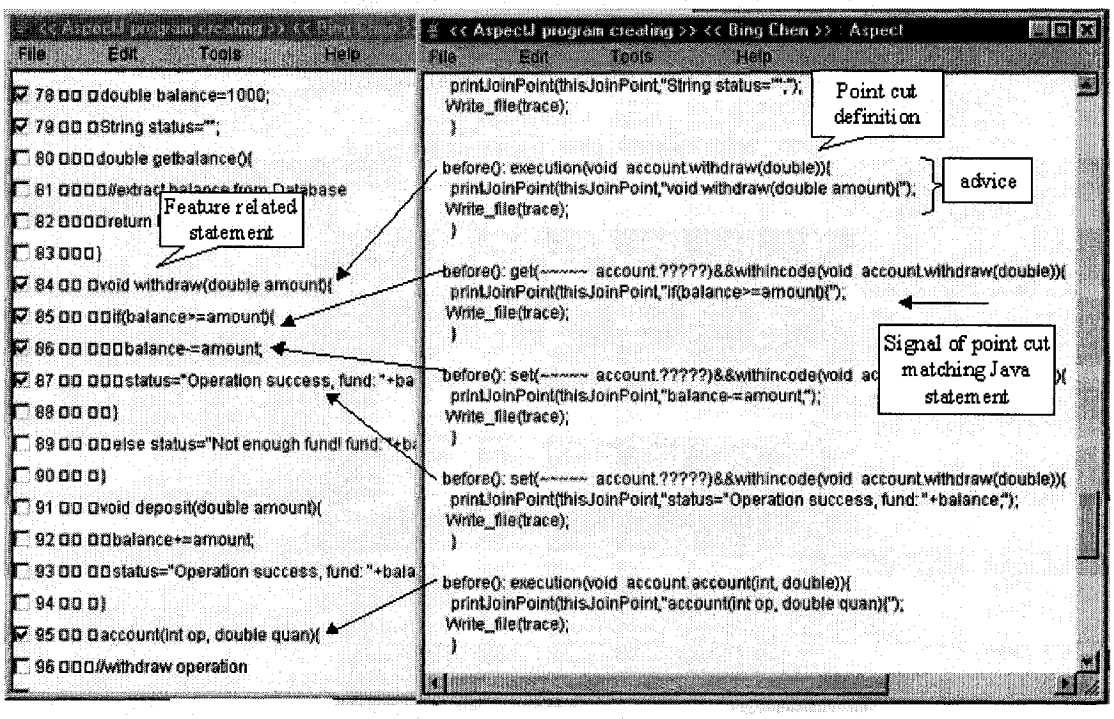



**Figure 4.2.3** Initialized AspectJ tracing program view

In Figure 4.2.3, the initialized AspectJ tracing program is shown in the right side panel. The *point cuts* match the *join points* (checked statements on the left side panel) one-to-one. The sequence of *point cuts* in tracing program is the same as the sequence of checked statements in source code.

Moreover, the *advice* content is constant in the AspectJ tracing program. Function "print_JoinPoint" extracts current statement information by using the AspectJ

*thisJoinPoint*. Another function "Write_file" writes the tracing information into a trace file.

The initialized tracing program still has some missing parts that require user interaction to complete, such as some missing parts in *point cuts*. This is due to the limitations of the parsers and semantic analysis. In addition, tracing program name, trace file name are also need user to define.

For example, in statement 86 ("balance -= amount;"), the *point cut* in initialized tracing program is:

```
set(~~~~~ account.?????)&&withincode(void account.withdraw(double)){
```

The *advice* is:

```
printJoinPoint(thisJoinPoint,"balance - = amount;");
Write_file(trace);
```

AspectJTracer provides two help options for users to complete the missing parts in the *point cuts*. The "Syntax Help" provides syntax mapping rule from Java statements to *point cuts* definition. The "Syntax Option" contains a list of possible solutions or guidelines to complete the missing parts in *point cuts*, which is shown in Figure 4.2.4.

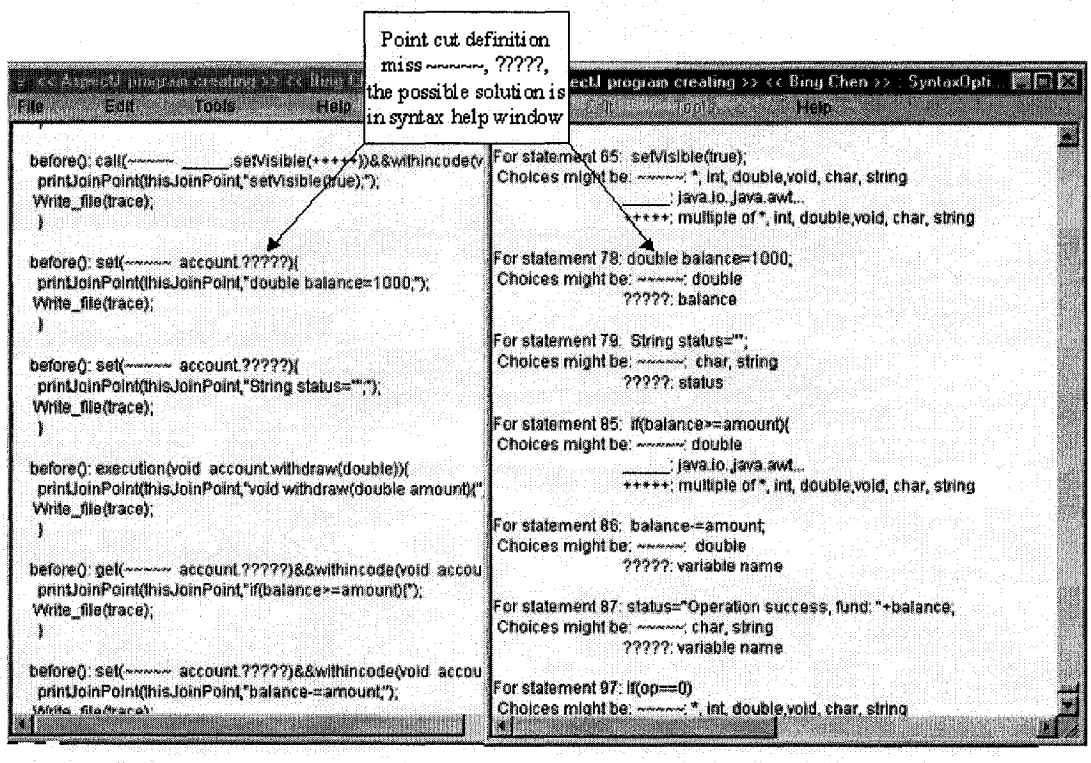



**Figure 4.2.4** Syntax option view

For statement 86 ("balance -= amount;" ), the *point cut* is:

```
set(~~~~~  account.?????) && withincode (void  account.withdraw()){
```

For the missing parts indicated by "~~~~~", possible answer in "Syntax Option" is "double". The place holder "?????" in this case corresponds to a missing variable name, user gets "balance" from "Syntax Option" window. As mentioned earlier, in this case the user will have to manually identify and fulfill this missing information. Therefore, the *point cut* for statement 86 is:

```
set(double  account.balance) && withincode(void  account.withdraw(double)){
```

The detailed AspectJ tracing program is attached in Appendix B.

The error checking for AspectJ tracing program is performed as part of the *weaving* process of the AspectJ tracing program with original Java program.

## 4.3 Result of Case Study

The AspectJ tracing program and Java program are weaved and executed to generate the execution trace, which is stored in a trace file. The trace file captures the execution of a functional feature and therefore supports the analysis, debugging and comprehension of the program. The execution trace of the first feature is in Appendix C.

The trace information stored in the trace file includes statement execution sequence number, statement content, parameter value, file name, and statement location.

## 4.4 Benefits of Current Source Code Instrumentation Approach

The advantages of the source code instrumentation approach with AspectJ are concluded as below:

*1. Improve source code instrumentation with AspectJ tracing functionality*

AspectJ has been adopted to improve source code instrumentation as AspectJ has a powerful tracing facility (section 2.5.5).

*2. Non-intrusive methodology*

The source code instrumentation approach outweighs traditional approaches in providing a non-intrusive method for Java source code. Source code does not need being modified to fit source code instrumentation. The feature of maintaining original Java source code provides the extensibility of the other new approach and research.

## 4.5 Limitations and Assumptions

The presented approach has certain assumptions with respect to the availability of software and domain knowledge. The following limitations are identified for the source code instrumentation approach:

*1. Not all statements are traceable in this approach*

Some statements, such as initialization of class, break, and continue, are not traceable in AspectJ syntax.

*2. Space requirements*

Each feature has an individual tracing program, thus the approach needs space to store these programs. Moreover, the trace result file also needs space to store.

*3. AspectJ knowledge*

Users of this software tool need to possess certain AspectJ programming language knowledge. AspectJTracer is a semi-automatic software tool of creating AspectJ

tracing program, and users need manually complete some parts in the tracing program, so they must have certain AspectJ programming language knowledge.

*4. Software requirements*

The process of debugging and compiling AspectJ tracing program and Java program requires user to possess AspectJ and Java integration environment, such as *Eclipse* and AspectJ API package.

# 5. Conclusions and Future Work

## 5.1 Conclusions

One of the major motivations of this research was to introduce a source code instrumentation approach that traces Java program with AspectJ tracing facility to represent functional features, which are extracted from source code with feature analysis techniques. Program comprehension, program slicing, program instrumentation, feature analysis, and AspectJ are reviewed in this thesis.

Source code instrumentation utilizing AspectJ can facilitate feature comprehension by capturing feature executions in *aspects*. A case study was implemented to validate the source code instrumentation experimentally.

This approach provides a non-intrusive method. However, limitations for this approach still exist, such as, not all statements are traceable, it has certain storage space requirements, the implementation of approach need command certain AspectJ knowledge, and it has specific software requirements.

## 5.2 Future work

Further research and improvements can be done in following fields:

- **Improve practicability of AspectJTracer**

  The software tool-AspectJTracer can not produce the tracing program automatically. However, AspectJ is a new programming language, and it is still in developing stage. It can be anticipated that in the future AspectJ is powerful enough to map more Java statement types to AspectJ *point cut*, which can improve the practicability of the software tool.

  Furthermore, an improved semantic parser will allow reduce the required user interaction during the mapping process by providing more detailed semantic information about the original Java source code.

- **Improve case study, GUI**

  A more complex and large scale case study may be applied in future to test the software tool efficiency, and GUI of this software tool can also be improved.

- **Better integration of parser, AspectJTracer, and programming environment.**

  The software tool-AspectJTracer took advantage of CONCEPT Parser, and was developed under *Eclipse* 3.0.1 environment. In the future, these systems can be integrated further for user to gain a more efficient usage.

# References

[Andr2001] Andrew Malton, The Software Migration Barbell, First ASERC Workshop on Software Architecture, August, 2001.

[Bela2004] Abhijit Belapurkar, Use AOP to maintain legacy Java applications, DeveloperWorks, March, 2004.

[Brok1983] R. Brooks, Towards a Theory of the Comprehension of Computer Programs, International Journal of Man-Machine Studies, Volumn 18, pages: 543-554, 1983.

[Chen2000] Chen, K., Rajlich, V., Case Study of Feature Location Using Dependence Graph, In Proceedings of the 8th International Workshop on Program Comprehension, pages: 241-249, June, 2000.

[Clo2005]Clover, "Frequently Asked Questions", http://www.cenqua.com/clover/doc/faq.html, retrieved Sep, 2005.

[Dai2005] Daikon Invariant Detector User Manual, http://pag.csail.mit.edu/daikon/, MIT Program Analysis Group, Retrieved Sep, 2005.

[Greg2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin,Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Proceedings of the 15th European Conference on Object-oriented Programming, Volume 2072, pages: 327-353, Springer-Verlag, 2001.

[Ham2004] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge, A survey of trace exploration tools and techniques, In Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, pages: 42-55, Markham, Ontario, Canada, 2004.

[Har2001] M. Harman, and R. M. Hierons, An overview of program slicing, Software Focus, Volumn 2, Issue 3, pages: 85-92, 2001.

[Hor1988] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, Volume 23, pages: 35-46, Atlanta, GA, June, 1988.

[Java1996] Sun Microsystems, Inc, The Java Language Specification Chapter 19 Grammar, J. Gosling, B. Joy, and G. Steele. The Java Language Specification Addison Wesley, 1996.

[Kicz1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming, In Proceedings of the 11th European Conference on Object-oriented Programming, Volumn 1241, pages: 220-242, Springer-Verlag, 1997.

[Kni1998] C. Knight, Visualization for program comprehension: information and issues, Technical Report, Department of Computer Science, University of Durham, 1998.

[Kor1988] Korel, B., and Laski, J. Dynamic program slicing. Information Processing Letters, Volumn 29, Issue 3, pages:155-163, October, 1988.

[Kor1994] Korel, B. and Yalamanchili, S. Forward Computation of Dynamic Program Slices, in Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis, pages: 66-79, 1994.

[Kosc2000] Koschke, R., Atomic Architectural Component Recovery for Program Understanding and Evolution, Phd. Thesis, University of Stuttgart, 2000.

[Lee1997] H. B. Lee, and B. G. Zorn, BIT: A Tool for Instrumenting Java Bytecodes, In Proceedings of the 1997 USENIX Sympo- sium on Internet Technologies and Systems (USITS'97), pages: 73-83, Dec, 1997.

[Lin1999] Lindholm, Tim and Yellin, Frank, The Java™ Virtual Machine Specification, Second Edition, Sun Microsystems, Inc., California, 1999.

[Mahr2002] D. Mahrenholz, O. Spinczyk, and Wolfgang-Schr¨oder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++, In Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE, pages: 249-256, 2002.

[Mai2005] Thomas Maier, Komor Alexander von B¨ulow, Georg Farber, MetaC and its Use for Automated Source Code Instrumentation of C Programs for Real-Time Analysis, In Proceedings of Work-In-Progress Session of the 17th Euromicro Conference on Real-Time Systems, pages: 29-33, July, 2005.

[Mart1999] Martin Fowler, Improving the Design of Existing Code, The Addison-Wesley Object Technology Series, June, 1999.

[Mich1996] Michael L. Nelson, A Survey of Reverse Engineering and Program Comprehension, ODU CS 551-Software Engineering Survey, April, 19, 1996.

[Nich2002] Nicholas Lesiecki, Improve modularity with aspect-oriented programming, Java Technology Zone, IBM Developer Works, January 2002.

[Palo2002] Palo Alto Research Center, the AspectJ Team, The AspectJ™ Programming Guide, Zerox Corporation, 2002.
http://aspectj.org/doc/dist/progguide/index.html, retrieved March, 2005.

[Pawel2003] Pawel Slowikowski, Krzysztof Zieliński, Comparison Study of Aspect-oriented and Container Managed Security, AAOS 2003: Analysis of Aspect Oriented Software: workshop in conjunction with ECOOP, pages: 1-6, Darmstadt, Germany, July 21 to July 25, 2003.

[Ralph1993] Ralph E. Johnson, Refactoring and Aggregation, In Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software, pages: 264 – 278, Kanazawa, Japan, November, 1993.

[Ram2002] Ramnivas Laddad, I Want My AOP! Part 2: Learn AspectJ to better understand aspect-oriented programming, JavaWorld, March, 2002.

[Ril2002] Rilling, J., Seffah, A., Bouthlier, C., The CONCEPT project - applying source code analysis to reduce information complexity of static and dynamic visualization techniques, Visualizing Software for Understanding and Analysis, pages: 90 – 99, June, 2002.

[Rob2000] Robert J. Walker, Gail C. Murphy, Jeffrey Steinbok, and Martin P. Robillard, Efficient Mapping of Software System Traces to Architectural Views, In Proceedings of CASCON 2000, pages: 31-40, 2000.

[Shnd1979] B.Shneiderman and R.Mayer, Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results, Int'l J. Computer and Information Sciences, Volumn 8, Issue 3, pages: 219-238, 1979.

[Sch1995] Schoenig, S. and Ducass'e, M., A hybrid backward slicing algorithm producing executable slices for Prolog, In Proceedings of the 7th Workshop on Logic Programming Environments, Portland, pages: 41-48, December, 1995.

[Stor1999] M.-A.D. Storey, F.D. Fracchia, and H.A. Muller, Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration, Journal of Software Systems, Volumn 44, pages: 171-185, 1999.

[Sus2005] Susmita Haldar, Reverse engineering based domain analysis, Master thesis, Concordia University, 2005.

[Thom2001] Thomas Eisenbarth, Rainer Koschke, Daniel Simon, Aiding Program Comprehension by Static and Dynamic Feature Analysis, In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01), page: 602-611, 2001.

[Van2002] A. van Deursen and E. Visser. The Reengineering Wiki, In Proceedings 6th European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society, pages: 217-220, 2002.

[Wake2002] William C. Wake, Extreme Programming Explored, Addison Wesley, 2002.

[Wei1982] Weiser, M. Programmers use slices when debugging. Communications of the ACM, Volume 25, Issue 7, pages: 446 – 452, July, 1982.

[Wild1995] Wilde, N., and Scully, M.C., Software Reconnaissance: Mapping Program Features to Code, Journal of Software Maintenance: Research and Practice, Volumn 7, Issue 1, pages: 49-62, 1995.

# Appendix A

```
/* Created on Nov-2006
 * @author chen bing */

import java.awt.*;
import java.awt.event.*;

public class bank{

        public static void main( String args[] ){
                new frame_class();
                //other banking service...
        }
 }

//      start of frame_class
        class frame_class extends Frame{
                TextField xinput;
                CheckboxGroup operation;
                Checkbox withdraw, deposit;
                Panel panel1;
                Button Conf;
                Label result1,result2;
                account account1;
                public frame_class(){
                        //GUI design
                        setLayout(new BorderLayout());
                        panel1=new Panel();
                        operation=new CheckboxGroup();
                        withdraw=new Checkbox("Withdraw", operation, true);
                        deposit=new Checkbox("Deposit", operation, false);
                        panel1.add(withdraw);
                        panel1.add(deposit);
                        add(panel1, BorderLayout.NORTH);
                        add(new Label("Amount:"), BorderLayout.WEST);
                        xinput=new TextField("");
                        add(xinput, BorderLayout.CENTER);
                        Conf= new Button("Confirm");
                        add(Conf,BorderLayout.EAST);

                        //add action listener
                        addListener();
                        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
                        pack();
                        setVisible(true);
                 }

                void addListener(){
                        Conf.addActionListener(new ActionListener(){
                                public void actionPerformed(ActionEvent evt){ action();}
                        });
                }
                public void action(){
                        //Read the withdraw or deposit amount
```

```
                        double amount =Double.parseDouble(xinput.getText());
                        //detect operation: withdraw or deposit
                        if(withdraw.getState())
                                account1= new account(0, amount);
                        else if(deposit.getState())
                                account1= new account(1, amount);

                                //respond the operation result
                                result1=new Label(account1.status);
                                add(result1, BorderLayout.SOUTH);
                                pack();
                        setVisible(true);
                }

                //action for window
                public void processEvent(AWTEvent evt){
                        if(evt.getID()==WindowEvent.WINDOW_CLOSING)
                        System.exit(0);
                        }
                //other action...
        }//end of frame_class

//      start of account class
        class account{
                        double balance=1000;
                        String status="";
                        double getbalance(){
                                //extract balance from Database
                                return balance;
                        }
                        void withdraw(double amount){
                                if(balance>=amount){
                                        balance-=amount;
                                        status="Operation success, fund: "+balance;
                                }
                                else status="Not enough fund! fund: "+balance;
                        }
                        void deposit(double amount){
                                balance+=amount;
                                status="Operation success, fund: "+balance;
                        }
                        account(int op, double quan){
                        //withdraw operation
                                if(op==0)
                                        withdraw(quan);
                        //deposit operation
                                else if(op==1)
                                        deposit(quan);
                        }
        }//end of account
```

# Appendix B

```
/*
 * Created on 3-Dec-2005
 * @author chenbing
 * Window - Preferences - Java - Code Style - Code Templates
 */

import org.aspectj.lang.*;
import org.aspectj.lang.reflect.*;
import java.io.*;
import java.awt.*;

        aspect bank_aspect{

         before(): execution(public void bank.main(String[])){
           printJoinPoint(thisJoinPoint,"public static void main( String args[] ){");
          Write_file(trace);
           }

          before(): call(frame_class.new())&&withincode(void  bank.main(String[])){
           printJoinPoint(thisJoinPoint,"new frame_class();");
           Write_file(trace);
           }

          before(): execution(frame_class.new()){
           printJoinPoint(thisJoinPoint,"public frame_class(){");
           Write_file(trace);
           }

          before(): call(* java.awt.Container.setLayout(..))&&withincode(frame_class.new()){
           printJoinPoint(thisJoinPoint,"setLayout(new BorderLayout());");
           Write_file(trace);
           }

          before(): call(Panel.new(..))&&withincode(frame_class.new()){
           printJoinPoint(thisJoinPoint,"panel1=new Panel();");
           Write_file(trace);
           }

          before(): call(CheckboxGroup.new(..))&&withincode(frame_class.new()){
           printJoinPoint(thisJoinPoint,"operation=new CheckboxGroup();");
           Write_file(trace);
           }

          before(): call(Checkbox.new(..))&&withincode(frame_class.new()){
           printJoinPoint(thisJoinPoint,"withdraw=new Checkbox(\"Withdraw\", operation,
true);/Deposit");
           Write_file(trace);
           }

          before(): call(* java.awt.Panel.add(..))&&withincode(frame_class.new()){
           printJoinPoint(thisJoinPoint,"panel1.add(deposit);/panel.add(withdraw);");
           Write_file(trace);
           }
```

```
before(): call(Label.new(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"add(new Label(\"Amount:\"), BorderLayout.WEST);");
Write_file(trace);
 }

before(): call(TextField.new(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"xinput=new TextField(\"\");");
Write_file(trace);
 }

before(): call(Button.new(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"Conf= new Button(\"Confirm\");");
Write_file(trace);
 }

before(): call(* java.awt.Component.add(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"add(Conf,BorderLayout.EAST);");
Write_file(trace);
 }

before(): call(void  frame_class.addListener())&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"addListener();");
Write_file(trace);
 }

before(): call(* java.awt.Component.enableEvents(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"enableEvents(AWTEvent.WINDOW_EVENT_MASK);");
Write_file(trace);
 }

before(): call(* java.awt.Window.pack(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"pack();");
Write_file(trace);
 }

before(): call(* java.awt.Component.setVisible(..))&&withincode(frame_class.new()){
 printJoinPoint(thisJoinPoint,"setVisible(true);");
Write_file(trace);
 }

before(): execution(void  frame_class.addListener()){
 printJoinPoint(thisJoinPoint,"void addListener(){");
Write_file(trace);
 }

before(): execution(void  frame_class.addActionListener()){
 printJoinPoint(thisJoinPoint,"Conf.addActionListener(new ActionListener(){");
Write_file(trace);
 }

before(): execution(void  frame_class.action()){
 printJoinPoint(thisJoinPoint,"public void action(){");
Write_file(trace);
 }
```

```
before(): call(double java.lang.Double.parseDouble(..))&&withincode(void
frame_class.action()){
 printJoinPoint(thisJoinPoint,"double amount =Double.parseDouble(xinput.getText());");
 Write_file(trace);
 }

before(): call(* java.awt.Checkbox.getState())&&withincode(void frame_class.action()){
 printJoinPoint(thisJoinPoint,"if(withdraw.getState())");
 Write_file(trace);
 }

before(): call(account.new(int, double))&&withincode(void frame_class.action()){
 printJoinPoint(thisJoinPoint,"account1= new account(0, amount);");
 Write_file(trace);
 }

before(): call(Label.new(..))&&withincode(void frame_class.action()){
 printJoinPoint(thisJoinPoint,"result1=new Label(account1.status);");
 Write_file(trace);
 }

before(): call(* java.awt.Component.add(..))&&withincode(void frame_class.action()){
 printJoinPoint(thisJoinPoint,"add(result1, BorderLayout.SOUTH);");
 Write_file(trace);
 }

before(): call(* java.awt.Window.pack(..))&&withincode(void frame_class.action()){
 printJoinPoint(thisJoinPoint,"pack();");
 Write_file(trace);
 }

before(): call(* java.awt.Component.setVisible(..))&&withincode(void frame_class.action()){
 printJoinPoint(thisJoinPoint,"setVisible(true);");
 Write_file(trace);
 }

before(): set(double account.balance){
 printJoinPoint(thisJoinPoint,"double balance=1000;");
 Write_file(trace);
 }

before(): set(String account.status){
 printJoinPoint(thisJoinPoint,"String status=\"\";");
 Write_file(trace);
 }

before(): execution(void account.withdraw(double)){
 printJoinPoint(thisJoinPoint,"void withdraw(double amount){");
 Write_file(trace);
 }

before(): get(double account.balance)&&withincode(void account.withdraw(double)){
 printJoinPoint(thisJoinPoint,"if(balance>=amount){");
 Write_file(trace);
 }
```

```
before(): set(double  account.balance)&&withincode(void account.withdraw(double)){
 printJoinPoint(thisJoinPoint,"balance-=amount;");
Write_file(trace);
 }

before(): set(String  account.status)&&withincode(void  account.withdraw(double)){
 printJoinPoint(thisJoinPoint,"status=\"Operation success, fund: \"+balance;");
Write_file(trace);
 }

before(): execution(void  account.account(int, double)){
 printJoinPoint(thisJoinPoint,"account(int op, double quan){");
Write_file(trace);
 }

before():  call(double  account.withdraw(double))&&withincode(void  account.account(int,
double)){
 printJoinPoint(thisJoinPoint,"withdraw(quan);");
Write_file(trace);
 }

    static int serialnumber=1;
    String trace=" The execution trace of feature: \n\n";
    String fileName = "feature1trace.txt";
    private void printJoinPoint(JoinPoint joinPoint, String source){
        trace+="Feature executing No: " + serialnumber +"   "+ source+"\n";
        StringBuffer argStr = new StringBuffer(" with Args:");
        Object[] args = joinPoint.getArgs();
        if (args.length == 0) argStr.append(" null");
            else
            for (int length = args.length, i=0;i<length;++i){
                argStr.append("["+i+"]="+args[i]);
                }
        SourceLocation sl=joinPoint.getSourceLocation();
        trace+=argStr;
        trace+="--Filename:"+sl.getFileName()+"--Source
location:"+sl.getLine()+"\n\n";
        serialnumber++;
        }
    private void Write_file(String str) {
        try {
            BufferedWriter out = new BufferedWriter(new FileWriter(fileName));
            out.write(str);
            out.close();
        } catch (IOException e) {
        System.out.println("IOException:");
        e.printStackTrace();
            }
        }
    }
```

# Appendix C

The execution trace of feature 1:

Feature executing No: 1 public static void main( String args[] ){
with Args:[0]=[Ljava.lang.String;@6eb38a--Filename:bank.java--Source location:9

Feature executing No: 2 new frame_class();
with Args: null--Filename:bank.java--Source location:10

Feature executing No: 3 public frame_class(){
with Args: null--Filename:bank.java--Source location:24

Feature executing No: 4 setLayout(new BorderLayout());
with Args: null--Filename:bank.java--Source location:26

Feature executing No: 5 panel1=new Panel();
with Args: null--Filename:bank.java--Source location:27

Feature executing No: 6 operation=new CheckboxGroup();
with Args: null--Filename:bank.java--Source location:28

Feature executing No: 7 withdraw=new Checkbox("Withdraw", operation, true);
With Args:[0]=Withdraw[1]=java.awt.CheckboxGroup[selectedCheckbox=null][2]=true--Filename:bank.java--Source location:29

Feature executing No: 8 deposit=new Checkbox("Deposit", operation, true);
with
Args:[0]=Deposit[1]=java.awt.CheckboxGroup[selectedCheckbox=java.awt.Checkbox[checkbox0,0,0,0x0,invalid,label=deposit,state=true]][2]=false--Filename:bank.java--Source location:30

Feature executing No: 9 panel1.add(withdraw);
with Args:[0]=java.awt.Checkbox[checkbox0,0,0,0x0,invalid,label=Withdraw,state=true]--Filename:bank.java--Source location:31

Feature executing No: 10 panel1.add(deposit);
with Args:[0]=java.awt.Checkbox[checkbox1,0,0,0x0,invalid,label=Deposit,state=false]--Filename:bank.java--Source location:32

Feature executing No: 11 add(panel1, BorderLayout.NORTH);
with Args: null:--Filename:bank.java--Source location:33

Feature executing No: 12 add(new Label("Amount:"), BorderLayout.WEST);
with Args:[0]=Amount:--Filename:bank.java--Source location:34

Feature executing No: 13 xinput=new TextField("");
with Args:[0]=--Filename:bank.java--Source location:35

Feature executing No: 14 add(xinput, BorderLayout.CENTER);
with Args: null:--Filename:bank.java--Source location:36

Feature executing No: 15 Conf= new Button("Confirm");
with Args:[0]=Confirm--Filename:bank.java--Source location:37

Feature executing No: 16 add(Conf,BorderLayout.EAST);

with Args: null:--Filename:bank.java--Source location:38

Feature executing No: 17 addListener();
with Args: null--Filename:bank.java--Source location:41

Feature executing No: 18 pack();
with Args: null--Filename:bank.java--Source location:43

Feature executing No: 19 setVisible(true);
with Args:[0]=true--Filename:bank.java--Source location:44

Feature executing No: 20 void addListener();
with Args: null--Filename:bank.java--Source location:47

Feature executing No: 21 public void action(){
with Args: null--Filename:bank.java--Source location:52

Feature executing No: 22 double amount =Double.parseDouble(xinput.getText());
with Args:[0]=65--Filename:bank.java--Source location:54

Feature executing No: 23 if(withdraw.getState())
with Args: null--Filename:bank.java--Source location:56

Feature executing No: 24 account1= new account(0, amount);
with Args:[0]=0[1]=65.0--Filename:bank.java--Source location:57

Feature executing No: 25 account(int op, double quan);
with Args:[0]=0[1]=65.0--Filename:bank.java--Source location:95

Feature executing No: 26 double balance=1000;
with Args:[0]=1000.0--Filename:bank.java--Source location:78

Feature executing No: 27 String status="";
with Args:[0]=--Filename:bank.java--Source location:79

Feature executing No: 28 withdraw(quan){
with Args:[0]=65.0--Filename:bank.java--Source location:98

Feature executing No: 29 void withdraw(double amount){
with Args:[0]=65.0--Filename:bank.java--Source location:84

Feature executing No: 30 if(balance>=amount){
with Args: null--Filename:bank.java--Source location:85

Feature executing No: 31 balance-=amount;
with Args:[0]=935.0--Filename:bank.java--Source location:86

Feature executing No: 32 status="Operation success, fund: "+balance;
with Args:[0]=Operation success, fund: 935.0--Filename:bank.java--Source location:87

Feature executing No: 33 result1=new Label(account1.status);
with Args:[0]=Operation success, fund: 935.0--Filename:bank.java--Source location:62

Feature executing No: 34 add(result1, BorderLayout.SOUTH);
with Args: null:--Filename:bank.java--Source location:63

Feature executing No: 35  pack();
 with Args: null--Filename:bank.java--Source location:64

Feature executing No: 36  setVisible(true);
 with Args:[0]=true--Filename:bank.java--Source location:65