

**INVESTIGATING THE RELATIONSHIP BETWEEN
ASPECTS AND PROGRAM SLICES**

By

YEE PING LU

Bachelor of Science in Computer Science

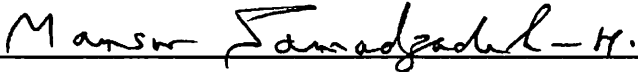
Oklahoma State University

2001

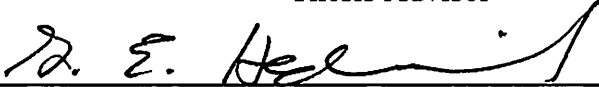
**Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 2004**

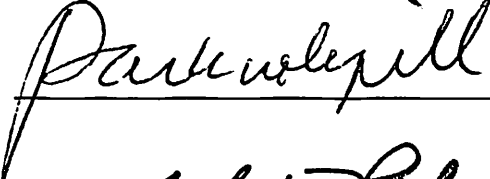
INVESTIGATING THE RELATIONSHIP BETWEEN
ASPECTS AND PROGRAM SLICES

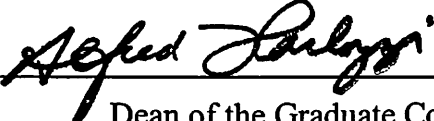
Thesis Approved:



Thesis Advisor







Dean of the Graduate College

PREFACE

Aspects are defined as system properties that crosscut components in a system's implementation. Aspect-oriented programming (AOP) is an approach based on aspects which is designed to handle complexities arising from crosscutting issues and aims at supporting the separation of concerns using aspects. Development of a software system comprises the design and implementation of the basic functionality as well as the system aspects such as synchronization, distribution, error handling, memory optimization, security management, exception handling, multi-object protocols, and resource sharing. Program slicing is a debugging and decomposition technique that extracts statements from a program relevant to a subset of its variables. The deleted part of the program does not affect the selected variables. Slicing reduces a program but still produces the behavior that the original program intended to produce with respect to a pre-specified subset of the variables.

This study was an investigation to better understand the relationship between aspects and program slices. Aspects crosscut a system based on the nature of each specific aspect under consideration. Program slices decompose a program according to a slicing criterion. Aspects deal with crosscutting issues in a program, while program slicing focuses on extracting the statements in a program that are relevant to a subset of the variables. Aspects and program slices have differences with respect to the identification criteria, application, decomposition, composition, tools support, targeted

languages, and theoretical basis. On the other hand, aspects and program slices seem to have similarity in testing, reusability, maintenance, and debugging. Testing as applied to aspects is slightly different from program slices because the programmer has to take care of special features in aspects such as joinpoints, pointcuts, and advices. Both aspects and program slices can be used to reduce the effort of debugging. For reuse, the concept of aspectual collaborations has been introduced, and slices have been used for identifying and isolating the reusable parts of programs.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor, Dr. Mansur H. Samadzadeh, for his supervision, guidance, assistance, and patience in completing my thesis work. I am especially appreciative of him for suggesting the topic for my research and his keen interest in my progress along the way. I am deeply impressed by his orderly and efficient way of working.

My sincere appreciation goes to my committee members, Drs. G. E. Hedrick and N. Park for their supervision and guidance.

Special thanks go to my parents in Malaysia for their sacrifices not to give up their enthusiasm to educate their children. I really appreciate all the support, encouragement, and love in the duration of my studies in the US.

Moreover, hearty thanks go to my best friend, Ken, for his love, patience, encouragement, and understanding during my up and down times.

Finally, I would like to thank the Computer Science Department of Oklahoma State University for its quality advanced education.

TABLE OF CONTENTS

Chapter	Page
I INTRODUCTION.....	1
II ASPECTS.....	3
2.1 Separation of Concerns.....	3
2.1.1 Crosscutting Concerns	4
2.1.2 Examples of Crosscutting Issues	5
2.2 Aspect Oriented Programming	8
2.2.1 Structure of AOP.....	11
2.3 Tools and Languages	12
2.3.1 Languages	12
2.3.2 Tool Support	13
2.3.3 AspectJ	13
2.3.4 Aspect Weaver	17
2.4 Sample Work and Metric	20
III PROGRAM SLICES	22
3.1 Static Slices.....	23
3.2 Dynamic Slices	24
3.3 Application of Program Slices	26
3.4 Tools Based on Slicing	27
IV JUXTAPOSITION	29
V SUMMARY AND FUTURE WORK.....	36
5.1 Summary	36
5.2 Future Work.....	37
REFERENCES	38
APPENDICES	43
APPENDIX A – GLOSSARY.....	43
APPENDIX B – TRADEMARK INFORMATION.....	45

LIST OF FIGURES

Figure	Page
1. Visualizing Crosscutting Concerns.....	4
2. Aspects Crosscut Classes in a Simple Figure Editor System	6
3. An Example of Crosscutting Concerns in a Distributed Digital Library	7
4. Examples of Crosscutting Concerns in a Record Store	8
5. Key Events in Program Execution at Joinpoints.....	15
6. A Pointcut Construct that Cut Across Multiple Classes	16
7. Construction of an Advice Takes Place After a Pointcut	16
8. A Simple Aspect Example with Pointcut and Advice	17
9. Programming Paradigms Based on Functional Decomposition vs. Aspect-Oriented Programming.....	18
10. Weaving Aspects and Components Together to Produce Overall Behavior	19
11. Sample Program with Static Slice.....	23
12. Sample Program with Static Slices.....	24
13. Sample Program with Dynamic Slice	25

LIST OF TABLES

Table	Page
I. Examples of Components and Aspects in Sample Domains.....	10
II. Dynamic Joinpoints of AspectJ.....	14
III. Juxtaposition of Aspects and Program Slices.....	34

CHAPTER I

INTRODUCTION

Object-oriented programming (OOP) can be considered a dominant programming practice. OOP is based on the idea of decomposing a system into objects and writing code for those objects. OOP attempts to achieve a clear separation of concerns at the source code level for constructing software systems. However, complex software systems have certain characteristics that hamper them from being cleanly and simply represented using the object-oriented approach. Such characteristics include different aspects of concern that typically crosscut the executable code such as synchronization, data storage, user interface, security, and error handling.

Aspects are defined as system properties that crosscut components in a system's implementation. Crosscutting occurs when two properties that are composed differently have to coordinate with each other. Aspect-oriented programming (AOP) makes it possible to clearly express the programs that OOP fails to support [Kiczales et al. 97].

Program slicing is a debugging and decomposition technique that focuses on a subset of the variables in a program and extracts those statements from the program that can impact the values of the selected subset of the variables at a certain point in the program. The deleted part of the program will not affect the selected variables. Slicing

reduces a program's size in general and produces the behavior that the original program intended to produce with respect to a pre-specified subset of the variables.

This thesis work explored the relationship between aspects and program slices. Aspects and program slices were investigated to find the similarities and the differences between them.

The rest of the thesis is organized as follows. Chapter II discusses the concept of aspects. It contains a discussion of separation of concerns, aspect-oriented programming, and tools and languages. Chapter III presents a brief overview of program slices including static slicing, dynamic slicing, applications of slices, and tools based on slicing techniques. Aspects and program slices are juxtaposed comparatively in Chapter IV. Chapter V presents the summary and future work on aspects and program slices.

CHAPTER II

ASPECTS

2.1 Separation of Concerns

Separation of concerns is an important software engineering principle. Separation of concerns refers to the ability to identify, encapsulate, and manipulate only the parts of software that are relevant to a particular concept, goal, or purpose [Ossher and Tarr 01].

The existing programming languages deal with the issue of separation of concerns by creating and explicitly calling subprograms. However, most of the time, a call to a subroutine is not enough to neatly or fully express separation of concerns. In order for the subprograms to function properly, both knowledge and cooperation is required on the part of the programmers of the calling components. The object-oriented model offers some capabilities for handling separation of concerns, yet it still has difficulty localizing concerns that do not fit naturally into a single program module or several closely related program modules. Due to the limitations of OOP, some design decisions that deal with separation of concerns cannot be illustrated with the object-oriented model [Kiczales et al. 97].

2.1.1 Crosscutting Concerns

An aspect is an area of concern that crosscuts the structure of a program (Figure 1). Some of the examples are data storage, user interface, platform-specific code, security, distribution, logging procedure, class structure, and threading. Most programming languages require programmers to make decisions about the implementation at the design stage. If programmers think that the design decisions might tangle the code, they might choose to, say, break the abstract classes in order to make the resulting parts reusable for other environments. Such a decision will cause a lot of overhead in the design and implementation process because redundancy will be the eventual outcome.

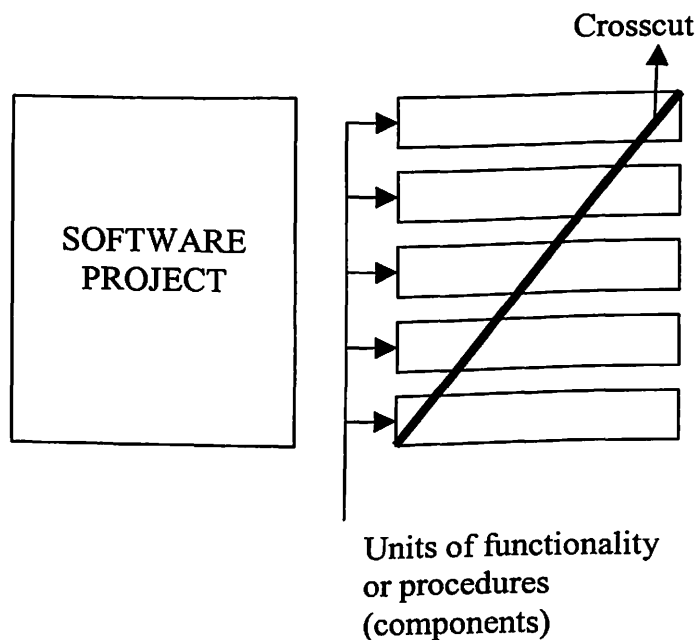


Figure 1. Visualizing crosscutting concerns

Current solution to overcome the crosscutting concerns is that the core concerns must be transformed to fit the solution for the problem. Therefore, the crosscutting phenomena are typically directly responsible for the tangling or unexpected complexity of the code. Code tangling results in harder code reuse, lower productivity, lower code quality, and limited evolution of the system.

2.1.2 Examples of Crosscutting Issues

Using a number of examples, this section explains how crosscutting issues occur in current programming paradigms. The first example is a simple figure editor system [AspectJ 02] [Elrad et al. 01b]. Consider two classes with clear and well-defined interfaces: Point and Line. A Figure consists of a number of FigureElements which can be Points or Lines. Whenever a FigureElement moves, it should notify the screen manager. This requires every method that moves a FigureElement to do the notification. Figure 2 shows that DisplayUpdating does not fit in either the Point box or the Line box, instead it cuts across both boxes. Using OOP, the implementation of the crosscutting concerns tends to be scattered across a system. However, by using AOP, the implementation can modularize the DisplayUpdating behavior into a single aspect [Elrad et al. 01b].

What follows is another example of crosscutting concerns in a distributed digital library. The graph shown in (Figure 3) is the ER (entity relationship) diagram of the database of a distributed digital library. The dotted line is where the crosscutting concern is. Whenever an activity occurs in one of the entities, the other entities have to be

notified. Each entity is referred to as a table. As depicted in the diagram, a user access cuts across the object structure [AspectJ 02].

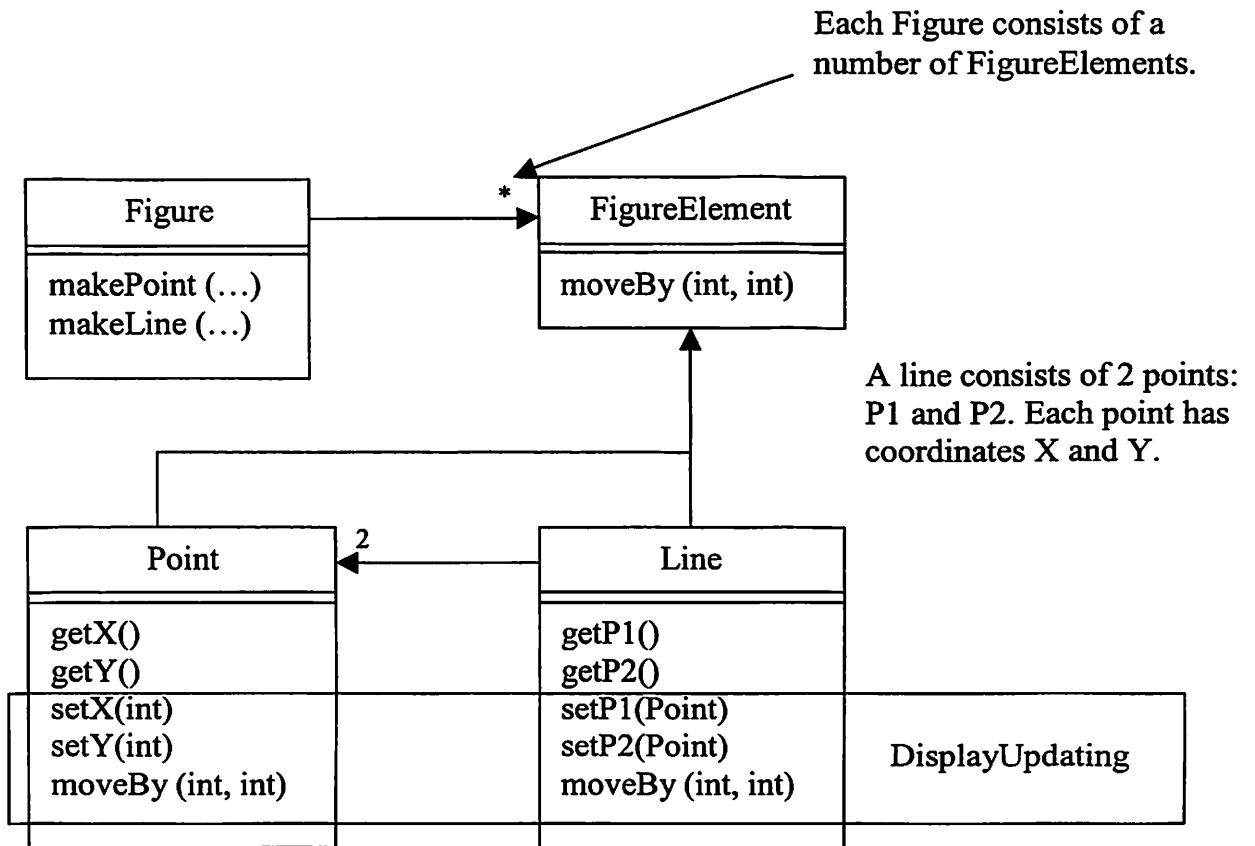


Figure 2. Aspects crosscut classes in a simple figure editor system [AspectJ 02]

The Library table holds many Documents and many Users. The Users access the Terminal by logging onto the computer at each terminal. Each logon on the Terminal's *logon ()* will affect the quota of *quota (user)* in the Library's table because Library is accessed by many Terminals. Each search on the Terminal's *search (key)* will be recorded in Library's table too via *search (key)*. Every printing job requested by the Terminal's *print (doc)* will be recorded in Library's table as *print (doc)*, and the request will be send to the Printer's table via *print (ps)*.

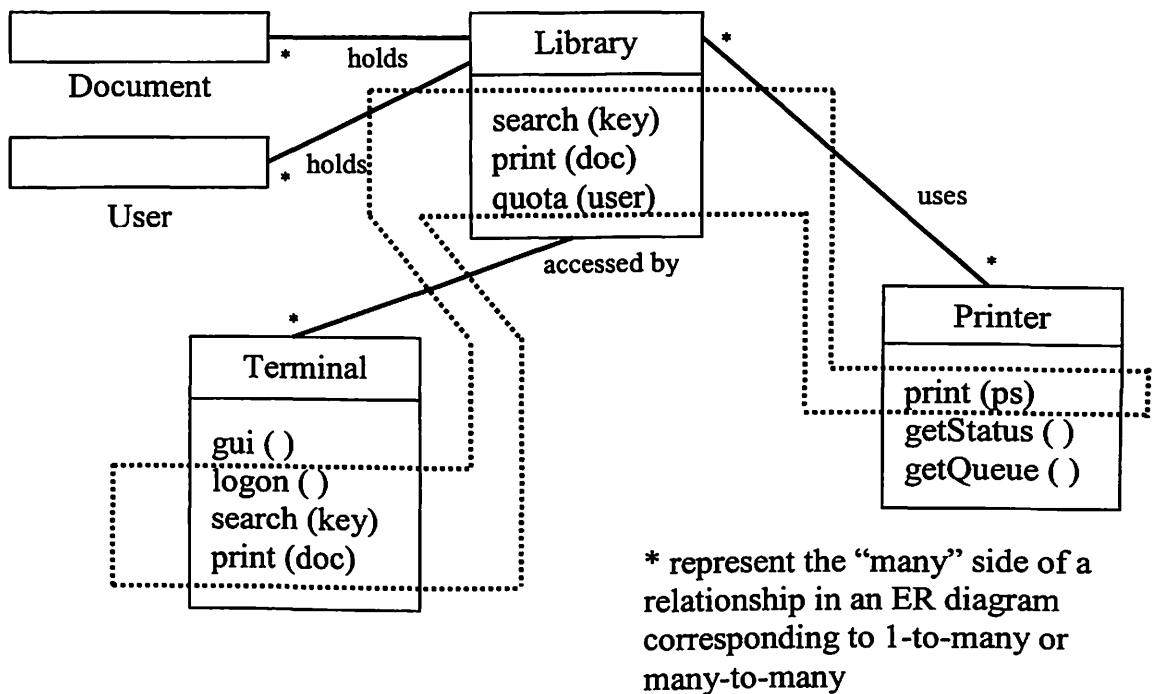


Figure 3. An example of crosscutting concerns in a distributed digital library [AspectJ 02]

For another example of crosscutting concerns, let's consider a record store selling CDs and magazines (Figure 4). Each CDs has its own title, artist, and label, while each magazines has title, author, and publication. The notion of "packaged item" cuts across both CDs and magazines because each item can be stored and retrieved, keeping track of where the CDs and magazines are located. The concern of "commodity" cuts across both CDs and magazines to keep track of the amount of CDs or magazines left before reordering their respective stocks. The company can get the sales amount from the "payment" concern which cuts across both of CDs and magazines. Whenever a CD and magazine is sold, it will affect `retrieve()`, `sell()`, and `charge()` for both CDs and magazines, and hence it will change the "packaged item", "commodity", and "payment" concerns.

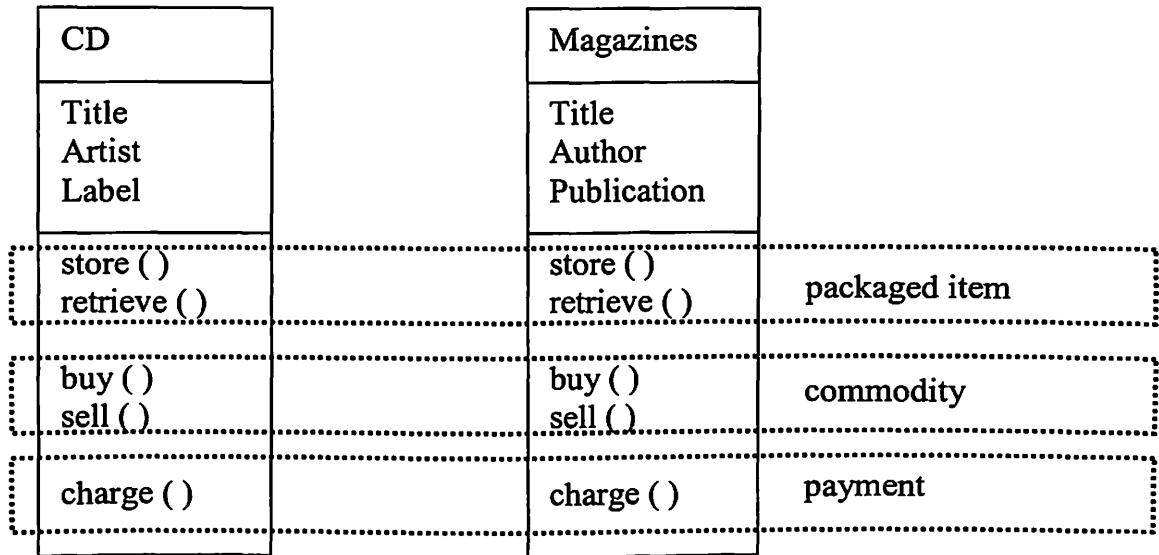


Figure 4. Examples of crosscutting concerns in a record store: packaged item, commodity, payment

2.2 Aspect Oriented Programming

Aspect-oriented programming (AOP) is in its early stages of existence. AOP is an approach designed to handle the complexities arising from crosscutting issues and aims at supporting the notion of separation of concerns using aspects. A major problem with the existing programming methods is that they are not generally sufficient to clearly capture some of the important design decisions. Kiczales and his colleagues used the term aspect to refer to the design decisions that are difficult to capture clearly in the actual code [Kiczales et al. 97]. Aspects are different from objects. Aspects can observe objects and react to their behavior. AOP allows a programmer to deal with design decisions separately by creating an aspect (or a set of aspects) for each area of concern.

Development of a software system comprises the design and implementation of the basic functionality required as well as capturing system aspects such as

synchronization, distribution, error handling, memory optimization, security management, exception handling, multi-object protocols, and resource sharing [Mehner and Wagner 99] [Kiczales and Hilsdale 01]. Existing conventional decomposition approaches target only the design and implementation modules, with aspects spread over the system and tangled with the code that captures the basic functionality of the system, thus making the system generally hard to develop, understand, and maintain.

Kiczales compares aspects to components using the following definitions [Kiczales et al. 97]. The term generalized procedure (GP) language is used to refer to the existing programming languages including object-oriented languages, procedural languages, and functional languages. A component is something that can be cleanly encapsulated in generalized procedure language, e.g., an object, a method, or a procedure. Cleanly mean well-localized or easily accessed, and composed as necessary. Components tend to be the units of a system's functional decomposition such as image filters, bank accounts, or GUI widgets.

An aspect is something that cannot be cleanly encapsulated in generalized procedure language. Aspects tend not to be the units of a system's functional decomposition, but rather properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects.

A number of examples are given below in Table I to differentiate between components and aspects [Kiczales et al. 97]. Error and failure handling are the most common aspects in almost all domains. The different failures that may occur and how a

failure should be handled crosscut the functionality of systems. Most of the performance-related issues are aspects too [Kiczales et al. 97].

Application Area	Generalized-Procedure Language	Components	Aspects
image processing	procedural	filters	loop fusion
			result sharing
			compile-time memory allocation
digital library	object-oriented	repositories	minimizing network traffic
		printers	synchronization constraints
		services	failure handling
matrix algorithms	procedural	linear algebra operations	matrix representation
			permutation
			floating point error

Table I. Examples of components and aspects in sample domains [Kiczales et al. 97]

The goal of AOP, based on the definitions provided by Kiczales and his colleagues [Kiczales et al. 97], is to separate components and aspects from each other cleanly by providing mechanisms that make it possible to abstract and compose them to produce the overall system.

Elrad and his colleagues [Elrad et al. 01] offered the following justification for aspect-oriented programming.

AOP is based on the idea that computer systems are better programmed by separately specifying the various concerns (properties or areas of interest) of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program.

2.2.1 Structure of AOP

AOP uses aspectual decomposition to break large problems down [Kiczales et al. 97]. In aspectual decomposition, the crosscutting and common concern are identified and then they are used to compose a problem into aspects. Aspectual decomposition enable developers to reason and to program using the natural aspects of concern of a system, even when those aspects crosscut both one another and the resulting executable code [Kiczales et al. 97]. In AOP, then each concern is implemented separately. In the last step of AOP, each concern or aspect is recomposed through a weaving process in which the source program is spread out and mixed in with other aspects in the output of the weaver.

AOP languages use five main elements to modularize the crosscutting concerns [Kiczales et al. 01b]:

- Joinpoints
- A means of identifying joinpoints (pointcuts)
- A means of specifying the behavior at joinpoints (advice)
- Encapsulated units combining joinpoints specifications (pointcuts) and behavior enhancements (advice)
- A method of attachment of units to a program (weaving)

See Section 2.3.3 for detailed explanations of these items.

2.3 Tools and Languages

Software design processes and programming languages work together to produce an overall system. Design processes break a system into smaller pieces and use programming languages to combine them together. To design an AOP system, the designer must clearly differentiate between the content of the component language, the content of the aspect language, and what must be shared between the two languages [Kiczales et al. 97]. A component language must allow a programmer to write component programs that implement a system's functionality and cannot preempt anything that the aspect programs may need to control. An aspect language must support the implementation of the desired aspects. Component and aspect languages are different most of the time, but they must have something in common to make it possible for the weaver (see Section 2.3.4) to co-compose the different kinds of programs.

2.3.1 Languages

Aspect languages belong to the language paradigm used in AOP. These languages which must support the separation of concerns for aspects [Mehner and Wagner 99]. Aspect languages work together with a base language for the basic functionality of a system. The base languages are such general languages as C++ or Java. AOP is a concept, so it is not bound to a specific programming language. AOP has been implemented in many languages using different base languages such as C, C++, C#, Perl, and Squeak/Smalltalk.

2.3.2 Tool Support

The most documented and used tool for AOP is AspectJ, an AOP implementation in Java (refer Section 2.3.3 for detailed information). Beside AspectJ, Hyper/J, AspectWerkz and JMangler are tools that support AOP in Java.

Hyper/J is a tool that supports flexible “multi-dimensional” separation, and the integration of concerns in standard Java software [Ossher and Tarr 00]. It is available free of charge on IBM’s alphaWorks [HyperJ 03].

AspectWerkz is a dynamic lightweight and high-performance AOP/AOSD framework for Java. It offers both power and simplicity for integration of AOP in both new and existing projects. It is free software available at <http://aspectwerkz.codehaus.org> [AspectWerkz 99].

JMangler is a framework for generic interception and transformation of Java programs at load-time [Kniesel et al. 01]. It allows one to change third-party Java classes without the source code. It is freely available under the terms of the GNU General Public License and partially under the terms of Sun Community Source License [JMangler 01].

2.3.3 AspectJ

AspectJ is a general-purpose and freely available aspect-oriented extension to Java [AspectJ 02]. AspectJ, which was developed by Xerox Palo Alto Research Center, enables the plug-and-play implementation of crosscutting concerns in Java. AspectJ was first prototyped in 1997 and released for public use in 1998. Version 1.0 of AspectJ was released in 2001. The latest version of AspectJ, Version 1.1, came out in June 2003. The

AspectJ project is sponsored by Palo Alto Research Center, NIST Advance Technology Program, and Defense Advanced Research Projects Agency. IBM and Eclipse are vendors that are supporting AspectJ too [Kiczales 03] [AspectJ 02].

AspectJ’s language construction extends the Java language so that every Java program is also a valid AspectJ program. In AspectJ, an aspect is declared by the keyword “aspect” and is defined in terms of joinpoints, pointcuts, and advices. Joinpoints are well defined points in a program’s execution that can be found in the source code by the AspectJ compiler. There are eleven types of joinpoints that AspectJ can possibly detect [Kiczales et al. 01b]. These eleven joinpoints are listed in Table II below.

<i>type of joinpoint</i>	<i>points in program execution at which...</i>
method call constructor call	a method (or constructor of a class) is called (call joinpoints are in the calling object, or in no object if the call is from a static method)
method call reception constructor call reception	an object receives a method or constructor call (reception joinpoints are before method or constructor dispatch, i.e., they happen inside a called object at a point in the control flow after control has been transferred to the called object, but before any particular method/constructor has been 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	the static initializers for a class, if any, are run
object initialization	the dynamic initializers for a class, if any, are run during object creation

Table II. Dynamic joinpoints of AspectJ [Kiczales et al. 01b]

Pointcuts are collections of joinpoints and certain values at those joinpoints. Advices are special method-like mechanisms that are used to declare that certain piece of

code should execute at each of the joinpoints in a pointcut. An advice is a piece of code that is triggered when the run-time context at a joinpoint meets specific conditions and can manipulate the surrounding local state or cause global effects [Walker et al. 03]. There are three types of advices: before advice, after advice, and around advice. Additionally, there are two special cases for after advice: after returning and after throwing.

There is a special kind of interface that consists of joinpoints existing between aspects and modules. These joinpoints are places in the base code that can be augmented by additional behavior and thus specified in an aspect [Mehner and Wagner 99].

Figure 5 shows the key events in program execution at joinpoints. The types of *joinpoints* in the example below are: *method call* and *method execution*. The cases for *advice* are: *after returning* and *after throwing*.

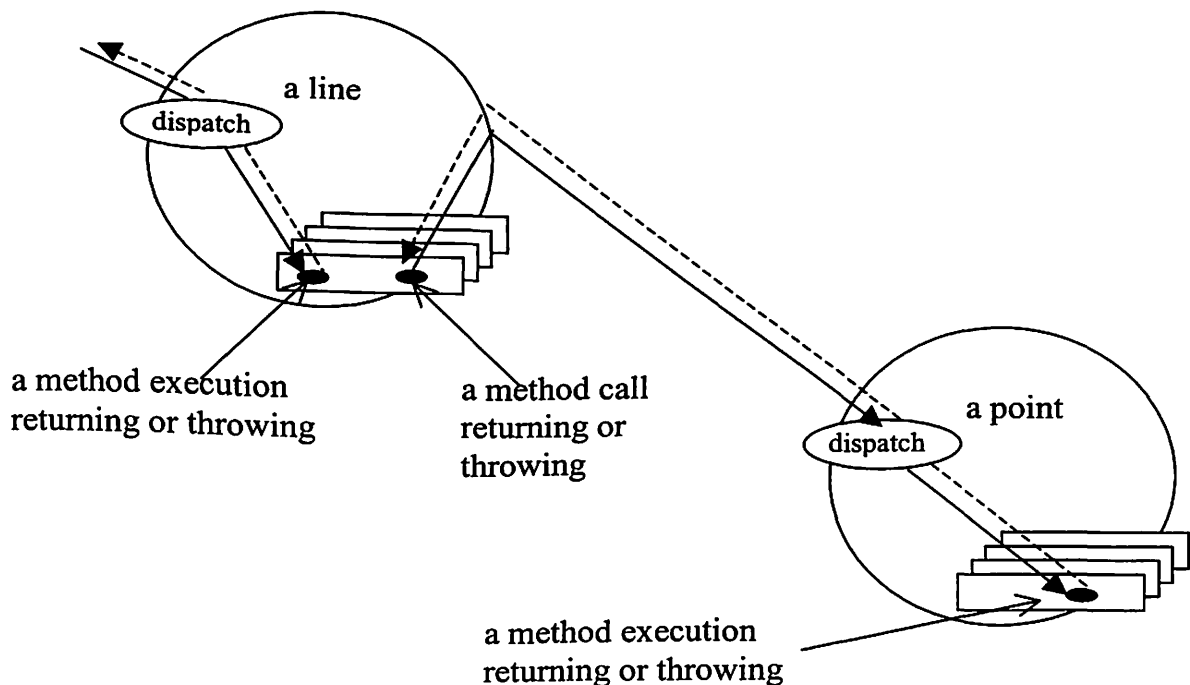


Figure 5. Key events in program execution at joinpoints [AspectJ 02]

Crosscutting issues can cut across multiple classes. For example in references to Figure 2 in Section 2.1.2, whenever a “point” receives void setX(int) or void setY(int) messages, or a “line” receives void setP1(Point) or void setP2(Point) messages, a crosscutting moves takes place. In Figure 6, *move* is the name of the pointcut and void Line.setP1(Point) is the method call.

```
pointcut move ( ):
    call (void Line.setP1(Point)) ||
    call (void Line.setP2(Point)) ||
    call (void Point.setX(int))    ||
    call (void Point.setY(int));
```

Figure 6. A pointcut construct that cut across multiple classes [AspectJ 02]

Advices are additional actions taken at crosscut. For example, in Figure 7, which is an extension from the previous example, additional action runs after the crosscut “move”.

```
pointcut move ( ):
    call (void Line.setP1(Point)) ||
    call (void Line.setP2(Point));

after ( ) returning: move ( ) {
    <code here runs after each move>
}
```

Figure 7. Construction of an advice takes place after a pointcut [AspectJ 02]

What follows is an example of a simple aspect with pointcut and advice [AspectJ 02]. The *aspect* here is *DisplayUpdating*. The *pointcut* is *move* and the type of *joinpoint*

is *method call*. The advice, which is the code that will run after a joinpoint is reached, is `Display.update()`.

```
aspect DisplayUpdating{
    pointcut move ( ):
        call (void Line.setP1(Point)) ||
        call (void Line.setP2(Point));

    after ( ) returning: move ( ) {
        Display.update( );
    }
}
```

Figure 8. A simple aspect example with pointcut “move” and after returning as an advice [AspectJ 02]

AspectJ enables clean modularization of crosscutting concerns such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols [AspectJ 02].

The weaver in AspectJ is a compiler that is apart from the regular compiler. AspectJ also introduces tools for debugging and documenting the code. The AspectJ compiler produces standard class files that follow the Java bytecode specification. The bytecode can then be interpreted on any compliant Java Virtual Machine (JVM).

2.3.4 Aspect Weaver

Aspect weaver is an important tool in aspect-oriented programming (AOP). Aspect weaver is used to combine or ‘weave’ an aspect code together with a program code before it is compiled into an executable module [Kiczales et al. 97] [Elrad et al. 01].

Aspect weaver processes both component and aspect languages and composes them properly to produce the desired total system operation. The concept of joinpoints (see Section 2.3.3 for a definition) is a must for aspect weaver because they are those elements of the component language semantics with which the aspect programs coordinate.

Figure 9 below shows the use of aspect weaver in aspect oriented programming and how it is different from functional decomposition programming. In programming paradigms based on functional decomposition (left), the code that is part of a functional unit in the source program remains relatively contiguous in the executable program. In aspect-oriented programming (right), the code that is part of the separate aspect descriptions in the source program is woven together and spread about in the executable program [Kiczales et al. 97].

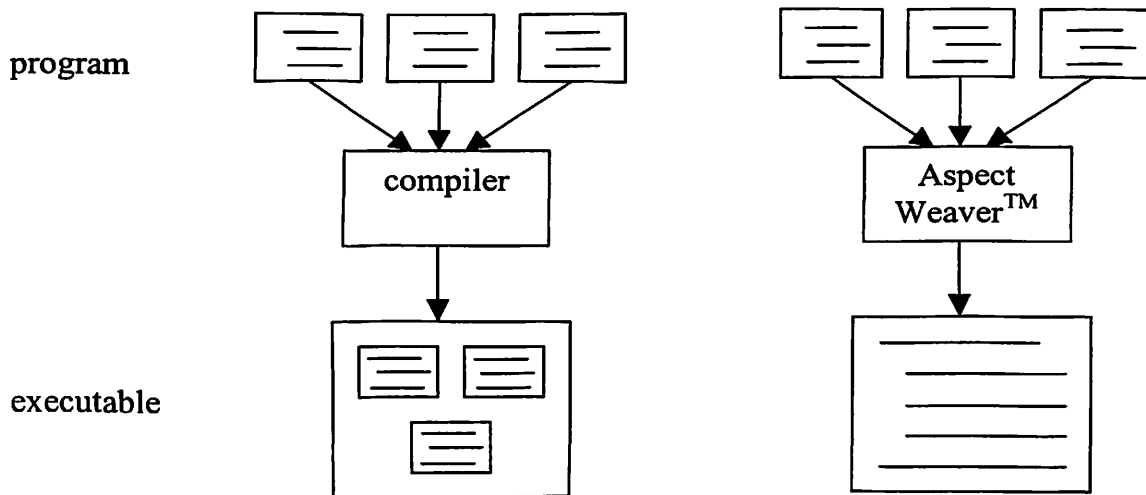


Figure 9. In programming paradigms based on functional decomposition (left), the code that is part of a functional unit in the source program remains relatively contiguous in the executable program. In Aspect-Oriented Programming (right), the code that is part of the separate aspect descriptions in the source program is woven together and spread about in the executable program [Kiczales et al. 97]

Figure 10 is the extended picture based on Figure 9 on aspect-oriented programming. In AOP, aspects and components are weaved together to produce the overall behavior. In the overall behavior, aspects and components are mixed together and they are not independent as before going through the Weaver [Constantinides et al. 00]. This is different from functional decomposition programming where the code that is part of a functional unit in the source program remains relatively contiguous in the executable program.

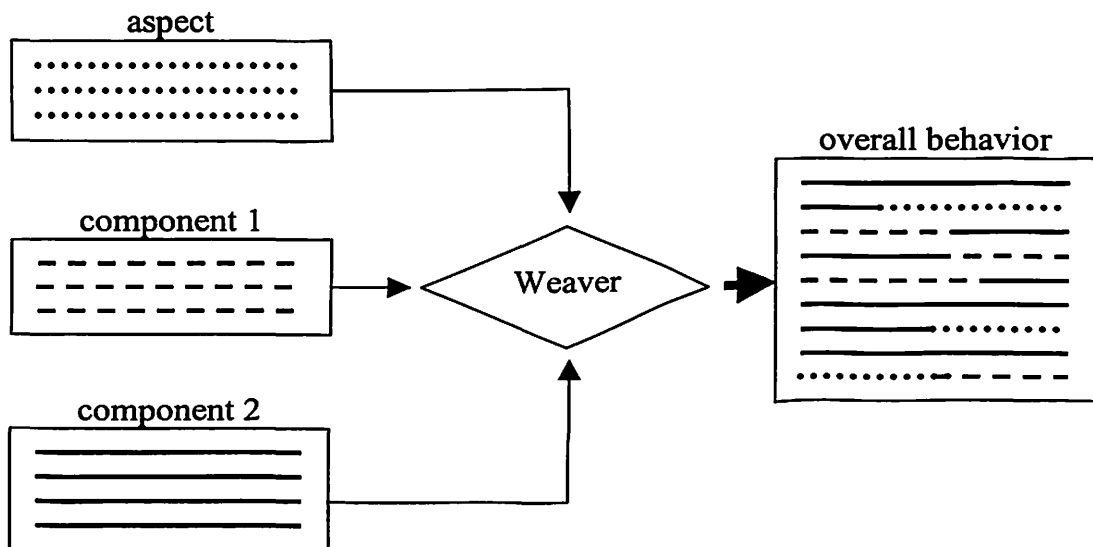


Figure 10. Aspects and components are weaved together and mixed around in the overall behavior and they are no longer independent source programs [Constantinides et al. 00]

2.4 Sample Work and Metric

Xerox Palo Alto Research Center created an example implementation of 768 lines of code and re-implemented it using tangled implementation, which does fusion optimization as well as memoization of intermediate results, compile-time memory allocation, and specialized intermediate data structures [Kiczales et al. 97]. As a result, they ended up with 35213 lines of code. This is extremely difficult to maintain since small changes to the functionality would require mentally untangling and then re-tangling the code.

AOP based re-implementation in the same sample work come out with:

- 1039 lines of code including the component program and three aspect programs,
- 3520 lines of aspect weaver code including a reusable code generation component, and
- 1959 lines for the true kernel of the weaver.

The general equation used for measurement suggested by Kiczales and his colleagues is given below [Kiczales et al. 97].

$$\text{reduction in bloat due to tangling} = \frac{\text{tangled code size} - \text{component program size}}{\text{sum of aspect program sizes}}$$

In the example mentioned at the beginning of this section, the measure compares the GP-based implementation of an application to an AOP-based implementation of the same application.

$$\text{reduction in bloat due to tangling} = \frac{\text{tangled code size} - \text{component program size}}{\text{sum of aspect program sizes}} = \frac{35213-756}{352} = 98$$

In this metric, any number greater than one indicates a positive outcome of applying AOP [Kiczales et al. 97]. This example indicates that by using AOP the programs can be easier to reason about, develop, and maintain at least for certain types of applications.

In addition to the example above, a lot of work has been done to explore AOP empirically. An experiment case study carried out by Murphy and his colleagues [Murphy et al. 01] showed that when locating faults within a single class or aspect, by using AspectJ, the programmers were able to correct a program fault faster than when using Java alone.

A study by Lippert and Lopes [Lippert and Lopes 00] found that implementations supported by AspectJ drastically reduced the portion of code related to exception detection and handling. The best case scenario in the study was when the code was reduced by a factor of four. The study showed that AspectJ also provided better support for different configurations of exceptional behaviors, more tolerance for changes in the specifications of exceptional behaviors, better support for incremental development, better reuse, automatic enforcement of contracts in applications that use the framework, and cleaner program texts.

Walker and his colleagues [Walker et al. 98] showed that AspectJ was able to complete the debugging tasks with fewer instances of semantic analyses. This seemed to lead directly to fewer instances of switching between files, indirectly to fewer builds, and ultimately to quicker completion times in the experiment.

CHAPTER III

PROGRAM SLICES

Program slicing is a decomposition technique that works by focusing on a subset of the variables of a program and extracting the relevant statements. It can be considered as a technique for simplifying programs by focusing on selected aspects of their semantics. The process of slicing deletes the parts of a program that can be determined to have no effect upon the semantics of interest. Slicing reduces a program to a minimal form while still producing its original behavior with respect to a subset of its variables. The reduced program, called a “slice”, is an independent program guaranteed to represent the original program within the domain of the specified subset of its behavior [Weiser 81].

A program slice consists of the parts of a program that potentially affect the values computed at some point of interest, referred to as a slicing criterion. Weiser defined a program slice S as a reduced, executable program obtained from a program P by removing statements, such that S replicates part of the behavior of P [Weiser]. A slice is also defined as a subset of the statements and control predicates of a program which directly or indirectly affect the values computed at a slicing criterion, but which do not necessarily constitute an executable program.

There are two types of slicing: static and dynamic, as explained in the following two sections.

3.1 Static Slicing

Static slicing refers to the slicing methods that preserve the behavior of a program for all possible executions without making any assumptions regarding the inputs [Weiser 81]. A static slice consists of a subset of program statements that affect a set of variables at a particular location in the program for all input combinations.

A slice is taken with respect to a slicing criterion $C = \langle s, \nu \rangle$, which specifies a location statement s in program P and ν is a subset of variables in P [Binkley and Gallagher 96]. For statement s and variable set ν , the slice of program P with respect to the slicing criterion $\langle s, \nu \rangle$ consists of all statements in the program that possibly affect the values of the variable set ν at s . A static slice includes all statements that affect variable set ν for all possible inputs at the point of interest. Two examples of static slicing appear below in Figures 11 and 12.

<pre>(1) input (n); (2) i := 1; (3) sum := 0; (4) product := 1; (5) while i <= n do begin (6) sum := sum + i; (7) product := product * i; (8) i := i + 1; end; (9) output (sum); (10) output (product)</pre> <p style="text-align: center;">(a)</p>	<pre>(1) input (n); (2) i := 1; (3) (4) product := 1; (5) while i <= n do begin (6) (7) product := product * i; (8) i := i + 1; end; (9) (10) output (product)</pre> <p style="text-align: center;">(b)</p>
--	--

Figure 11. (a) An example program. (b) A static slice with respect to the slicing criterion $\langle s, \nu \rangle$ where $s = 10$ and $\nu = \{\text{product}\}$ [Tip 94]

There are two directions for computing slices: backward slices and forward slices.

A backward slice traverse a program back to its beginning starting at the slicing criterion, while a forward slice traverse dependences in the forward direction [Tip 94].

<pre>(1) begin (2) read (X, Y) (3) total := 0.0 (4) sum := 0.0 (5) if X <= 1 (6) then sum := Y (7) else begin (8) read (Z) (9) total := X*Y (10) end (11) write (total, sum) (12) end</pre>	<pre>(1) begin (2) read (X, Y) (5) if X < 1 (6) then (7) else (8) read(Z) (12) end</pre>
	<pre>(1) begin (2) read (X, Y) (12) end</pre>
	<pre>(1) begin (2) read (X, Y) (3) total := 0.0 (5) if X <= 1 (6) then (7) else (9) total :=X*Y (12) end</pre>
(a)	(b)

Figure 12. (a) Sample program. (b) Static slices with respect to the slicing criteria (12, {Z}), (9, {X}) and (12, {total}) [Weiser 81]

3.2 Dynamic Slicing

Korel and Laski first introduced dynamic program slicing in 1988 [Tip 94] [Binkley and Gallagher 96]. A dynamic program slice is the part of a program that affects the computation of a variable or variables of interest during program execution on

a specific program input [Agrawal and Horgan 90]. A dynamic slice is taken with respect to a slicing criterion $\langle i, v, s \rangle$ which consists of a set of variables v , input i , and from the beginning to statement s in the program. Dynamic program slicing refers to a collection of program slicing methods that are based on program execution and may significantly reduce the size of a static program slice because run-time information collected during program execution is used to compute dynamic program slices.

<pre> (1) input (n) (2) i := 1; (3) while (i <= n) do begin (4) if (i mod 2= 0) then (5) x := 17 else (6) x:= 18; (7) i := i +1; end; (8) output (x); </pre> <p style="text-align: center;">(a)</p>	<pre> (1) input (n) (2) i := 1; (3) while (i <= n) do begin (4) if (i mod 2= 0) then (5) x := 17 else (6) i := i +1; end; (8) output (x); </pre> <p style="text-align: center;">(b)</p>
--	---

Figure 13. (a) An example program. (b) A dynamic slice with respect to criterion $\langle i, v, s \rangle$ which is $(n=2, \{x\}, 8^1)$ in this case [Tip 94].

In Figure 13 above, the input value is $n=2$ and therefore the program loops twice. The variable set is $\{x\}$. The value of x can be 17 or 18, but since $s = 8^1$, only the first occurrence of the program is needed. Thus, $x := 18$ is not needed in the dynamic slice. If this program is sliced using static slicing, it will consist of the entire program. This example shows that dynamic slicing is more sensitive to certain inputs compared to static slicing with respect to the size of the resulting slice.

3.3 Application of Program Slicing

Slices are used by programmers during debugging because they potentially allow a programmer to ignore a large number of statements in the process of localizing a bug, especially when a programmer is confronted with a large program. Slicing and slices can be used for program verification, program integration, program comprehension, software maintenance, testing, debugging, software quality assurance, and reengineering [Tip 94] [Binkley and Gallagher 96] [Agrawal and Horgan 90].

An example of the use of slicing for debugging follows. If a program computes an incorrect value for variable x , only the statements included in the slice with respect to x could have possibly contributed to the error, therefore all statements which are not in the slice can safely be ignored. Static slicing methods can help isolate the code containing the erroneous statement(s). Static slicing can be used to locate the error in a program caused by uninitialized variables that are used in expressions. In dynamic slicing, a “slice” consists only of the statements that influence the value of a variable for specific program inputs. Therefore, dynamic slicing is better suited to assist the programmer in locating a bug for a particular execution of a program.

Program slicing can be used by software maintainers to make changes to software without having a negative impact on the unchanged part. A relatively new kind of slice, called a decomposition slice, has proven useful in making a change to a piece of software without unwanted side effects [Binkley and Gallagher 96]. For example, when a variable v is determined to have a value to be changed, the program is partitioned into three parts, independent, dependent, and compliment. The statement in the independent slice with respect to v are not in any other decomposition slice. On the other hand, the statement in

the dependent slice with respect to ν do exist in other decomposition slices. Compliment slices contain statements that may exist in some other decomposition slices, but not in the slices for ν .

In reuse engineering, program slicing can be used to isolate code fragments thus implementing reusable functional abstractions. Program slicing has been used both for structural and specification driven method. A relatively new slicing process called Specification Driven Program Slicing has been recently introduced [Chung et al. 01]. The thesis report will not go into the details of Specification Driven Program Slicing, as the purpose here is to compare and contrast slices with aspects.

3.4 Tools Based on Slicing

There are several tools based on slicing that have been introduced either as research prototypes or as commercial products. Oberon Slicing Tool (OST) is published under the Oberon Slicing Tool License. Copyright of OST belongs to Christoph Steindl [Steindl 99]. Some of the features of OST are listed below.

- It computes programs slices of Oberon-2 programs with no restrictions.
- It appears to be very efficient and computes slices within a few seconds.
- It uses static slicing techniques which produce potentially larger slices than dynamic slicing, but the calculation of the static slices are more efficient.
- It can handle both procedural and object-oriented programs in an expression-oriented way.
- It uses user-feedback to restrict the effects of aliases and dynamic binding.

- It uses a repository to store the computed slicing information which can be re-used later when importing already sliced modules.

Unravel is a prototype static program slicing tool from National Institute of Standards and Technology (NIST) funded by both the United States Nuclear Regulatory Commission (NRC) and the National Communications System (NCS) [Unravel 99]. Unravel can be used to evaluate ANSI C source code statically. It can identify code that is executed in more than one computation by combining program slices with logical set operations. It is commonly used when dealing with safety system and security.

CodeSurfer is a commercial program slicing tool [GramaTech 03]. Copyright of CodeSurfer belongs to GrammaTech. CodeSurfer supports programs written in C the language and can be used to look for programming errors because the user can select any statement in the program.

Samadzadeh and Wichaipanitch [Samadzadeh and Wichaipanitch 93] developed a debugging tool for C programs, called C-debug, based on dynamic slicing and dicing techniques. Wichaipanitch [Wichaipanitch 03] developed an interactive debugging tool for C++ programs based on dynamic slicing and dicing.

CHAPTER IV

JUXTAPOSITION

In Chapters II and III, aspects and program slices were described in detail. In this chapter, the notions of aspects and slices are compared and contrasted. A number of selected observations about aspects and program slices are explained in this chapter. There are some commonalities as well as some differences between aspect and program slices that are juxtaposed in this chapter. The goal is to achieve a clear understanding of the relationship between aspect and program slices.

Program slicing is a relatively mature area of research compared to research in aspects which is in its beginning stages. Program slicing has been studied primarily in the context of procedural programming languages [Zhao 02c]. With the presence of aspect-oriented programming, a new area of research has been opened in program slicing, which is the slicing of aspect-oriented software. Slicing reduces a program to a minimal form while still producing its original behavior with respect to a subset of its variables. The existing slicing algorithms for conventional procedural languages or object-oriented languages cannot be applied straightforwardly to aspect-oriented programming. The reason is specific features such as aspects, joinpoints, pointcuts, and advices in aspect-oriented programming. Therefore, the slicing of aspect-oriented programs must be handled in a different way. Aspect-oriented system dependence graph has been

introduced as an extension to dependence graph that is used in program slicing. Zhao presented a detailed study of the slicing of aspect-oriented software [Zhao 02c].

Applications of program slicing in software engineering activities include testing, debugging, maintenance, reverse engineering, and complexity measurement. In program slicing, a program satisfies a conventional data flow testing criterion if all def-use pairs occur in a successful test case [Binkley and Gallagher 96]. Data flow testing is defined as testing values which associate with the variables that can effect the execution of a program [Zhao 02d]. Data flow testing in aspects and several support tools for unit testing of aspect-oriented software is proposed by Zhao [Zhao 02a] [Zhao 02d]. The types of testing introduced by Zhao for aspects are intra-module testing, inter-module testing, and intra-aspect testing. Intra-module testing is the testing of an individual module in the aspect such as a piece of advice, an introduction, or a method. Inter-module testing is performed on a public module along with some other modules that it calls directly or indirectly in an aspect, but it doesn't involve any other module outside the aspect. Intra-aspect testing is performed on the interactions of multiple public modules in an aspect when they are called in a random sequence from the outside of the aspect [Zhao 02a] [Zhao 02d].

The basic testing unit in AOP is an aspect. An aspect is designed to work as independently as possible from its environment. This allows a programmer to write small test programs to exercise a particular aspect in a program [Zhao 02d]. The small testing program written by the programmer can be weaved to the program base code to test the program, and it can be discarded after testing is done.

A program composed of aspects consists of two parts: *non-aspect code*, which includes some classes, interfaces, and other language constructs as in Java, and *aspect code*, which includes aspects for modeling the crosscutting concerns in the program. On the other hand, program slices are subsets of the original program, which means the program slices are parts of the original program's source code.

Program slicing research was originally motivated by debugging. Program slices ease the debugging process because they slice out the irrelevant parts of the code and let the programmer to continue from a particular slice of code. On the other hand, a study by Walker and his colleagues [Walker et al. 98] shows that by applying AOP to the debugging task, one can achieve better results in shorter time. By applying program slicing to aspect-oriented programming, there is greater chance to reduce the effort of the debugging process because it allows the users to ignore many program statements that are not relevant to the erroneous code in an aspect-oriented program [Zhao 01]. AOP is used to modularize a program's ability for collecting dynamic information for program analysis. Therefore, reducing the effort of the debugging process is an important step towards efficient software evolution.

Program slices are used mostly for maintenance purposes. Software maintainers generally would like to try to make changes to software without having a negative or unexpected impact on the unchanged parts of the software. A new kind of slice, called a decomposition slice, is useful in making a change to a piece of software without unwanted side effects [Binkley and Gallagher 96]. An aspect can be used for maintenance by applying change impact analysis which is explained in next paragraph. Due to the "crosscutting" feature of aspects, an aspect can easily cut across a certain part

of the code in the program. An aspect code can be written separately and weaved into the program for maintenance without making changes throughout the whole program.

Change impact analysis allows one to capture the change effect information of the software so that one can perform software evolution actions on aspect-oriented software [Zhao 02b]. It provides the necessary techniques to address the problems by identifying the changes of the software and uses the information gathered to re-engineer the software system design. Zhao proposed an approach to support change impact analysis of aspect-oriented software based on program slicing techniques [Zhao 02b]. The main feature of the approach is to estimate the effect of making the changes in an aspect-oriented program by analyzing its source code and automatically completing the process of change impact analysis [Zhao 02b]. Applying program slicing to support change impact analysis in aspect-oriented software is beneficial for software evolution in aspect-oriented programming. For example, when a programmer tries to make some changes in an aspect-oriented program, the programmer must first investigate which statements will affect or will be affected by the modified statement. A slicing tool can assist a programmer in change impact information by extracting the parts containing the statements that might affect, or be affected by, the modified statements. Thus, the programmer needs to examine only the statements included in the slices in order to investigate the impact of modification.

Both aspects and program slices can be used in software development with reuse. The concept of aspectual collaborations was introduced to enhance aspect reuse [AspectJ 02]. The AOP modularity technology should result in better, cleaner, and more reusable

code quickly and easily. As for slicing, it has long been used for identifying the reusable functions.

Aspects and program slices have differences with respect to the basic program unit, identification criteria, application, decomposition, composition, tools support, targeted languages, and theoretical basis. On the other hand, aspects and program slices has the similarity in program testing, debugging, maintenance, and reverse engineering. Selected observations about aspects and program slices have been placed side-by-side in Table III.

Table III. Juxtaposition of aspects and program slices

JUXTAPOSITION POINT	ASPECTS	PROGRAM SLICES
<i>Identification Criteria</i>	Identify aspects based on crosscutting issues.	Identify slices according to the given slicing criteria.
<i>Application</i>	Aspect-oriented programming is used to deal with the crosscutting issues and supports the separation of concerns.	Program slices are used to focus on a subset of the statements in a given program.
<i>Usage</i>	Primarily useful in the design stage to help developers in writing better modularized code based on separation of concerns.	Useful in debugging, testing, and maintenance of programs.
<i>Decomposition</i>	In AOP, a given design problem is decomposed into concerns that can be localized into separate modules and concerns that tend to crosscut over a set of modules.	Using various slicing criteria, portions of a code that correspond to certain behaviors of the program can be isolated.
<i>Composition</i>	Various aspects (properties or areas of interest) of a system can be separately specified along with some description of their relationships, and then, relying on mechanisms in the AOP environment, we can weave or compose them together into a coherent program.	Program slices can be grouped into a library for future composition. Slices stored in the library can be reused to compose new programs.
<i>Executable/Non-executable</i>	An aspect is not an executable program. An aspect has to be weaved into a base program to become executable.	A dynamic slice is an executable program whose behavior must be identical to the specified subset of the original program's behavior with respect to variables of interest at some execution position. A static slice is a subset of the statements and control predicates in a program which directly or indirectly affect the values computed at the criterion, but which do not necessarily constitute an executable program.

<i>Maintenance</i>	Aspects help modularize the implementation in that the code dealing with each particular area of concern is grouped together as a unit resulting in a more easily maintainable system.	Program slicing is used in maintenance by changing the source code without unwanted side effects.
<i>Reusability</i>	The concept of aspectual collaboration has been introduced to enhance aspect reuse. AOP modularity generally results in better, cleaner, and more reusable code.	Program slicing has been used for identifying reuse candidates in code and constructing repositories of potentially reusable components.
<i>Debugging</i>	Aspects ease the debugging process by designing a good set of trace points (the tracing is disabled when it is not being used) without making changes throughout the whole program.	Program slicing allows a programmer to focus on those statements in a given program that may have contributed to a fault.
<i>Tool Support</i>	Some examples of tool support are AspectJ, Hyper/J, AspectWerkz, and JMangler.	Some examples of tool support are Oberon Slicing Tool, Unravel, and Codesurfer.
<i>Targeted Languages</i>	Aspect-oriented programming mostly targets Java.	Program slicing targets conventional procedural languages and object-oriented languages.
<i>Theoretical Basis</i>	Flow graphs and formal definitions for Turing-complete crosscut languages.	Program dependence graphs, control flow graphs, spanning trees, and graph decomposition.
<i>Run-Time Information</i>	AOP languages and weavers can be designed so that weaving work is delayed until run-time.	Dynamic slicing uses run-time information for computing slices.
<i>Compile-Time Information</i>	Aspect of a system can be changed, inserted, and removed at compile time.	Static program slices is a compile-time program analysis.
<i>Modularization</i>	This is the main idea of AOP which solves the problems caused by crosscutting concerns.	Modularization based on slices, especially when constructing a system from slices, achieves the goals of error isolation and containment.
<i>Object-Orientation</i>	AOP complements OOP, where object-oriented languages are used to write base program and aspects offer the powerful addition of the ability to crosscut to OOP.	Program slicing has been applied to object-oriented programs. Oberon Slicing Tools has been developed for slicing object-oriented programs.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

Aspect-oriented programming (AOP) solves some of the problems that current programming paradigms are not able to solve. Although AOP does not solve all of the existing problems, it definitely improves the current status of handling the crosscutting concerns. AOP does not replace object-oriented programming (OOP), rather it builds on OOP by supporting the separation of concerns that OOP handles poorly. The ability to modularize the implementations of crosscutting concerns is the distinctive feature of AOP.

Program slicing research is quite a mature area. It is mainly used for debugging, testing, reuse, and maintenance. Application of AOP to program slices can lead program slicing research to a new direction.

The findings in this study are based on examining the concepts of aspects and slices, and the theory behind them. No actual implementation was done in this study. From the juxtaposition, criteria such as identification, decomposition, composition, application area, usage, tool support, targeted languages, theoretical basis, run-time information, and compile-time information show the differences between aspects and slices. On the other hand, both aspects and slices are applicable for maintenance, reuse,

testing and debugging. For testing and debugging, aspects and slices can probably be applied together.

5.2 Future Work

AOP seems to have a promising future. In all probability, aspects will not be the last word for handling separation of concerns in software development. There is room for future research and development in the area of aspects.

The current popular aspect language, AspectJ, is a noteworthy start. But AspectJ is only used on top of the Java programming language. What about other languages such as C, C++, Perl, and Fortran? Aspect language should be more generalized. By achieving this, the usage of aspect will not be restricted to just one language. Will aspect languages become independent languages instead of just being applicable to other languages?

From the reuse point of view, will aspects-based software development overcome component-based software development? Perhaps aspects can be reused like program slices, or even be placed in an aspect library.

Programmers spend around 70% of the time maintaining programs. Aspects and slices are both beneficial for maintenance. Qualification of the contribution of aspects to improve maintenance is a desirable goal.

REFERENCES

- [Agrawal and Horgan 90] Hiralal Agrawal and Joseph R. Horgan, "Dynamic Program Slicing", *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 246-256, White Plains, New York, June 1990.
- [AspectJ 02] AspectJ Official Website, AspectJ Crosscutting Objects for Better Modularity, <http://eclipse.org/aspectj/>, last modified: December 18, 2002, access date: October 18, 2003.
- [AspectWerkz 99] AspectWerkz Official Website, <http://aspectwerkz.codehaus.org/>, creation date: February 1999, access date: March 9, 2004.
- [Binkley and Gallager 96] David W. Binkley and Keith B. Gallager, "Program Slicing", In *Advances in Computers*, Vol. 43, Edited by: Marvin Zelkowitz, pp. 1-50, Academic Press, San Diego, California, 1996.
- [Chung et al. 01] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon, "Program Slicing Based on Specification", *Proceedings of the 2001 ACM Symposium on Applied Computing*, pp. 605-609, Las Vegas, Nevada, March 2001.
- [Constantinides et al. 00] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, Mohamed E. Fayad, and P. Netinant, "Designing an Aspect-Oriented Framework in an Object-Oriented Environment", *ACM Computing Surveys*, Vol. 32, No. 41, pp. 1-12, March 2000.
- [Elrad et al. 01a] Tzilla Elrad, Robert E. Filman, and Atef Bader, "Aspect-Oriented Programming", *Communication of the ACM*, Vol. 44, No.10, pp. 28-32, October 2001.
- [Elrad et al. 01b] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Liebeherr, and Harold Ossher, "Discussing Aspects of AOP", *Communications of the ACM*, Vol. 44, No.10, pp. 33-38, October 2001.
- [GrammaTech 01] GrammaTech CodeSurfer Software Analysis and Understanding Tool, <http://www.grammatech.com/products/codesurfer/overview.html>, last modified: September 21, 2001, access date: September 25, 2003.
- [HyperJ 03] HyperJ Official Website, <http://www.alphaworks.ibm.com/tech/hyperj>, last modified: July 8, 2003, access date: March 9, 2004.

- [JMangler 01] JMangler (Aspects Tool) Website, available online at <http://javalab.cs.uni-bonn.de/research/jmangler/>, Copyrights of ROOTS group, Computer Science Dept. III, University of Bonn, 2001.
- [Kiczales 03] Interview with Gregor Kiczales, "Aspect-Oriented Programming", TheServerSide.com J2EE Community Events Interview posted online at <http://www.theserverside.com/events/videos/GregorKiczalesText/interview.jsp>, interview date: July 2003, access date: September 11, 2003.
- [Kiczales et al. 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-Oriented Programming", *Proceedings of ECOOP'97 – 11th European Conference on Object-Oriented Programming*, pp. 220-242, Jyväskylä, Finland, June 1997.
- [Kiczales and Hilsdale 01] Gregor Kiczales and Erik Hilsdale, "Aspect-Oriented Programming", *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundation of Software Engineering*, Vol. 26, No. 5, pp. 313, Vienna, Austria, September 2001.
- [Kiczales et al. 01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mil Kersten, Jeffrey Palm, and William G. Griswold, "Getting Started with AspectJ", *Communications of the ACM*, Vol. 44, No. 10, pp. 59-95, October 2001.
- [Kiczales et al. 01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffery Palm, and William G. Griswold, "An Overview of AspectJ", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)*, pp. 327-353, Budapest, Hungary, June 2001.
- [Kniesel et al. 01] Günter Kniesel, Pascal Costanza, and Michael Austermann, "JMangler – A Framework for Load-Time Transformation of Java Class Files", *Proceedings of IEEE Workshop on Source Code Analysis and Manipulation*, pp. 100-110, Florence, Italy, November 2001.
- [Lesiecki 02] Nicholas Lesiecki, "Improve Modularity with Aspect-Oriented Programming", Technical Team Lead, eBlox, Inc., Published online at <http://www-106.ibm.com/developerworks/java/library/j-aspectj>, January 2002.
- [Lippert and Lopes 00] Martin Lippert and Cristina Videira Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming", *Proceedings of the 22nd International Conference on Software Engineering*, pp. 418-427, Limerick, Ireland, June 2000.
- [Lopes 97] Cristina V. Lopes, "D: A Language Framework for Distributed Programming", Ph.D. Thesis, Graduate School of the College of Computer Science, Northeastern University, Boston, Massachusetts, 1997.

- [Lopes and Kiczales 98] Cristina V. Lopes and Gregor Kiczales, "Recent Developments in AspectJ", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'98)*, pp. 398-401, Brussel, Belgium, July 1998.
- [Mehner and Wagner 99] K. Mehner and A. Wagner, "An Assessment of Aspect Language Design", a position paper published online at <http://www.uni-paderborn.de/cs/ag-engels/Papers/1999/MehnerYRW99.htm>, *Young Researchers Workshop, Generative and Component-Based Software Engineering (GCSE)'99*, Messe Erfurt, Germany, September 1999.
- [Mendhekar et al. 97] Anurag Mendhekar, Gregor Kiczales, and John Lamping, "RG: A Case-Study for Aspect-Oriented Programming", Technical report SPL97-009 P9710044, Xerox PARC, Palo Alto, California, February 1997.
- [Murphy et al. 01] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mil A. Kersten, "Does Aspect-Oriented Programming Work?", *Communications of the ACM*, Vol. 44, No. 10, pp. 75-77, October 2001.
- [Ossher and Tarr 00] Harold Ossher and Peri Tarr, "Hyper/J: Multi-Dimensional Separation of Concerns for Java," *22nd International Conference on Software Engineering*, pp. 734-747, Limerick, Ireland, June 2000.
- [Ossher and Tarr 01] Harold Ossher and Peri Tarr, "Using Multidimensional Separation of Concerns to (Re)shape Evolving Software", *Communications of the ACM*, Vol. 44, No. 10, pp. 43-50, October 2001.
- [Pace and Campo 01] J. Andres Diaz Pace and Marcelo R. Campo, "Analyzing the Role of Aspects in Software Design", *Communications of the ACM*, Vol. 44, No. 10, pp. 43-50, October 2001.
- [Samadzadeh and Wichaipanitch 93] M. H. Samadzadeh and W. Wichaipanitch, "An Interactive Debugging Tool for C Based on Dynamic Slicing and Dicing", *Proceedings of the Twenty-First Annual ACM Computer Science Conference (CSC '93)*, pp. 30-37, Edited by Stan C. Kwasny and John F. Buck, Indianapolis, Indiana, February 1993.
- [Samadzadeh et al. 00] M. H. Samadzadeh, A. Usman, and M. K. Zand, "Investigating the Relationship Between Threads and Program Slices as Reusable Software Units", *Proceedings of the Eighth International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU2000)*, pp. 1881-1888, Madrid, Spain, July 2000.
- [Steindl 99] Christoph Steindl, "The Oberon Slicing Tool", a demonstration paper published online at <http://ecoop99.di.fc.ul.pt/techprogramme/demonstrations.html>, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, Lisbon, Portugal, June 1999.

- [Tip 94] F. Tip, “A Survey of Program Slicing Techniques”, Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, Netherlands, 1994.
- [Unravel 99] The Unravel Program Slicing Tool, <http://www.nist.gov/itl/div897/sqg/unravel/unravel.html>, last modified: September 2, 1999, access date: September 25, 2003.
- [Voas and Viega 00] Jeffrey Voas and John Viega, “Can Aspect-Oriented Programming Lead to More Reliable Software?”, *IEEE Software*, Vol. 17, No. 6, pp. 19-21, November/December 2000.
- [Wand 03] Mitchell Wand, “Invited Talk – Understanding Aspects (Extended Abstract)”, *8th ACM SIGPLAN International Conference on Functional Programming*, pp. 299-300, Uppsala, Sweden, August 2003.
- [Walker et al. 98] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy, “Assessing Aspect-Oriented Programming: Preliminary Results”, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP’98)*, pp. 433-434, Brussels, Belgium, August 1998.
- [Walker et al. 99] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy, “An Initial Assessment of Aspect-oriented Programming”, *Proceedings of the 21st International Conference on Software Engineering*, pp.120-130, Los Angeles, California, May 1999.
- [Walker et al. 03] David Walker, Steve Zdancewic, and Jay Ligatti, “A Theory of Aspects”, *8th ACM SIGPLAN International Conference on Functional Programming*, pp. 127-139, Uppsala, Sweden, August 2003.
- [Weiser 81] Mark Weiser, “Program Slicing”, *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439-449, San Diego, California, March 1981.
- [Weiser 82] Mark Weiser, “Programmers Use Slices When Debugging”, *Communications of the ACM*, Vol. 25, No. 7, pp. 446-452, July 1982.
- [Wichaipanitch 03] Winai Wichaipanitch, “An Interactive Debugging Tool for C++ Based on Dynamic Slicing and Dicing”, Ph.D. Dissertation, Computer Science Department, Oklahoma State University, Stillwater, Oklahoma, May 2003.
- [Zhao 01] J. Zhao, “Dependence Analysis of Aspect-Oriented Software and Its Applications to Slicing, Testing, and Debugging”, Technical-Report SE-2001-134-17, Information Processing Society of Japan (IPSJ), October 2001.
- [Zhao 02a] J. Zhao, “Data Flow Testing of Aspects”, Technical Report SE-136-26, Information Processing Society Japan (IPSJ), March 2002.

- [Zhao 02b] J. Zhao, "Change Impact Analysis for Aspect-Oriented Software Evolution", *Proceedings of the Fifth International Workshop on Principles of Software Evolution*, pp. 108-112, Orlando, Florida, May 2002.
- [Zhao 02c] J. Zhao, "Slicing Aspect-Oriented Software", *Proceedings of the Tenth IEEE International Workshop on Program Comprehension*, pp. 251-260, Paris, France, June 2002.
- [Zhao 02d] J. Zhao, "Tool Support for Unit Testing of Aspect-Oriented Software", a position paper presented at OOPSLA'02, Object-Oriented Programming Systems, Languages, and Application, Seattle, Washington, November 2002, Workshop on Tools for Aspect-Oriented Software Development posted online at <http://www.cs.ubc.ca/~murphy/OOPSLA02-Tools-for-AOSD/>.

APPENDIX A

GLOSSARY

Advice	A piece of code that is triggered when the run-time context at a joinpoint meets specific conditions.
AOP	Aspect-Oriented Programming, a way of building information systems in which common domain-crossing design decisions are modularized in separate layers of code.
Aspect	A programming module that contains the implementation of a crosscutting concern.
AspectJ	A general-purpose and free aspect-oriented extension of Java which enables plug-and-play implementation of crosscutting concern in Java.
Aspect Language	A language that supports the implementation of desired aspects.
Aspect Weaver	The compiler in AspectJ that is used to weave the aspect code together with the program code before it is compiled into an executable program.
Component	A constituent part of a system.
Component Language	A language that allows a programmer to write component programs to implement a system's functionality using components.
Core Concern	The center of behavior of interest in an aspect.
Crosscutting Concerns	Design problems that exhibit themselves globally across functional modules and/or objects such as synchronization, distribution, error handling, memory optimization, security management, exception handling, multi-object protocols, and resource sharing.

Dynamic Slicing	Slicing performed by specifying values for input variables. A dynamic slicing criterion specifies an input together with a static slicing criterion.
GUI	Graphical User Interface.
Joinpoints	Well-defined execution points of a program that can be found in a source code by the AspectJ compiler, e.g, method calls, method executions, constructor calls, constructor executions, field references, and field assignments.
OOP	Object-Oriented Programming, a method of program implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, where the classes are all members of a hierarchy of classes united via an inheritance relationship.
Pointcuts	Collections of <i>joinpoints</i> and certain values at those joinpoints.
Program Slice	A subset of program statements obtained by <i>program slicing</i> based on a particular <i>slicing criterion</i> .
Program Slicing	A family of techniques involving operations on source code that isolate a part of the behavior of a program when viewed from a point of interest within the program.
Separation of Concerns	The ability to identify, encapsulate, and manipulate only the parts of a specification or a program that is relevant to a particular concept, goal, or purpose.
Slicing Criterion	Specification for a particular behavior of interest in a program. For static slicing, the slice of a program P with respect to the slicing criterion $\langle s, v \rangle$ includes those statements of P needed to capture the behavior of the program from its beginning to statement s for variable v. For dynamic slicing, the dynamic slice of a program P with respect to the dynamic slicing criterion $\langle i, v, s \rangle$ is the statements of P from its beginning to statement s for variable v when the input to the program is i.
Static Slicing	Slicing performed without considering the input values.

APPENDIX B

TRADEMARK INFORMATION

AspectJ	Trademark of Palo Alto Research Center, Inc.
CodeSurfer	Trademark of GrammaTech, Inc.
Java	Trademark of Sun Microsystems, Inc.

VITA #1

Yee Ping Lu

Candidate for the Degree of

Master of Science

Thesis: INVESTIGATING THE RELATIONSHIP BETWEEN ASPECTS AND PROGRAM SLICES

Major Field: Computer Science

Biographical:

Personal Data: Born in Sarikei, Sarawak, Malaysia, February 6, 1981, daughter of Choon Woo Lu and Sui Nguk Ling.

Education: Graduated from Sarikei City Government High School, Sarawak, Malaysia, in December 1997; received Bachelor of Science degree in Computer Science from the Computer Science Department at Oklahoma State University, Stillwater, Oklahoma, US, in December 2001; completed the requirements for Master of Science degree in Computer Science at the Computer Science Department of Oklahoma State University in May 2004.

Experience: Employed by Computing and Information Services (CIS), currently known as Information Technology Department (ITD), Oklahoma State University, as a Computer Lab Assistant from October 2000 to December 2003.