INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

NOTE TO USERS

This reproduction is the best copy available.

UMI

Class Firewalls in Java

by

Vu Nguyen

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science
Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

1999

© copyright

1999, Vu Nguyen

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*

Our file *Notre référence*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-48439-4

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis

Class Firewalls in Java

submitted by

Vu Nguyen

in partial fulfillment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

June 1999

Abstract

Regression testing is an important activity in software maintenance. Unfortunately, regression testing is also an expensive software activity. Ideally, it should be applied only to those parts that are affected by the software modifications. Software change impact analysis is the study of identifying the consequences of changes that are introduced to a software system. Thus, it is an essential step in an efficient regression testing strategy. Class Firewall theory, an OO impact analysis approach, attempts to locate all classes affected by changes made to a class, based on well-defined dependency relationships within classes.

This thesis presents a tool, named JFireClass, which automatically reverse-engineers Java software to construct class firewalls for modified classes. The underlying idea in computing the class firewall is based on the transitivity of mutual dependencies that exist among Java classes. By testing only those affected classes, testers can significantly reduce their test efforts. The proposed tool also produces a test order, which serves as a sequencing guideline for regression testing. In essence, the test order establishes a sequence in which components can be tested from previously tested components, which leads to a reduced test effort. In addition, this thesis also proposes some techniques to handle the dependencies missing due to polymorphism. To demonstrate JFireClass's correctness and usefulness, JGL (Java Generic Language – a commercial package) was used as the target for impact analysis.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Jean-Pierre Corriveau for supporting me in my efforts to complete this interesting subject; in particular, his valuable suggestions and technical review of the paper. I owe a substantial debt of gratitude to Robert Beaudry, my supervisor at Statistics Canada, who has given me all the possible facilities to complete the work and revised every section of the paper. I am also grateful to my parents, who desire to see their son soaring high, for their constant encouragement; in particular to my dad, Phong-Chau Nguyen, for his tirelessly reading and correcting the paper day or night. Last but not least, I continue to be no less grateful to my wife, Chau N.G. Pham, for her patience, understanding and support me for over the last two years.

Vu Nguyen

Carleton University
Ottawa, Ontario, Canada
June, 1999

Table of Contents

Abstract	iii
Acknowledgements	iv
Table of Contents	v
List of Figures	viii
1. Chapter 1- Introduction	1
1.1. Introduction	1
1.2. Motivations	7
1.3. General Contributions	8
1.4. Outline of the Thesis	9
2. Chapter 2 – Background	11
2.1. Object-Oriented Concepts	11
2.2. Classes and Objects Relationships	14
2.2.1. <i>Inheritance Relationships</i>	14
2.2.2. <i>Interface Relationships</i>	15
2.2.3. <i>Aggregation (Whole-Part) Relationships</i>	16
2.2.4. <i>Association (Use) Relationships</i>	16
2.3. Java as an OO language	17
2.3.1. <i>Java in General</i>	17
2.3.2. <i>Inner Classes</i>	18
2.4. OO Regression Testing and Software Change Impact Analysis	22
2.4.1. <i>Regression Testing</i>	23
2.4.2. <i>Software Change Impact Analysis</i>	25
2.4.2.1. Past Work on Dependency Analysis	28
3. Chapter 3 - Class Firewall in Java	32
3.1. Identifying Class Relationships in Java	33
3.1.1. <i>Inheritance Relationships</i>	34
3.1.1.1. Class Inheritance Relationships	34
3.1.1.2. Polymorphic Class Inheritance Relationships	34
3.1.1.3. Interface Inheritance Relationships	34
3.1.2. <i>Aggregation Relationships</i>	35
3.1.2.1. Static Aggregations	35
3.1.2.2. Automatic Aggregations	35
3.1.2.3. Implicit Inner-Class Aggregations (IIC)	36
3.1.3. <i>Association Relationships</i>	37
3.1.3.1. Local Variables	37
3.1.3.2. Object Parameter Passing	37
3.1.3.3. Static Variables	38
3.1.3.4. Local Classes - Anonymous Classes	38
3.1.4. <i>Interface Relationships</i>	39
3.1.4.1. Strict-Data Interface Relationships	40
3.2. Object Relation Graph (ORG)	40
3.3. Class Firewall in Java	42
3.3.1. <i>Class Firewall Theory</i>	44
3.3.2. <i>Discussions on the Implementation of the Class Firewall Algorithm</i>	48
3.4. Testing Order	52
3.4.1. <i>Overview of Kung's Test Order Algorithm [Kung et. al., 95a]</i>	54

3.4.2. <i>Converting a Cyclic ORG to an Acyclic ORG</i> <i>Using Clusters [Kung et. al., 95a]</i>	55
3.4.3. <i>Cycle Breaking</i>	55
3.4.4. <i>Analysis on the Complexity of Kung's Test Order Algorithm</i>	56
3.5. <i>An Alternative of the Test Order Algorithm</i>	56
3.5.1. <i>Overview of the Alternative Test Order Algorithm</i>	58
3.5.1.1. <i>An Example of how Topological Sorting Produces a Test Order</i>	59
3.5.2. <i>Cycle Breaking in Java</i>	60
3.5.2.1. <i>Interface Relationship Does Not Involve in Cycles</i>	60
3.5.2.2. <i>Cycle Breaking Rule in Java</i>	61
3.5.3. <i>The Cycle Breaking Algorithm</i>	63
3.5.4. <i>Analysis of the Correctness of the Alternative Test Order Algorithm</i>	64
3.5.5. <i>Analysis of the Complexity of the Alternative Algorithm</i>	65
3.6. <i>Discussion on the Effectiveness of the Class Firewall Strategy</i>	66
 4. Chapter 4 - Detailed Examples	67
4.1. <i>UML Notation in JFireClass</i>	68
4.1.1. <i>Class Structure</i>	68
4.1.2. <i>Class Relationships</i>	69
4.2. <i>Bank Example</i>	69
4.2.1. <i>The Class Diagram and the Corresponding ORG of the Bank Example</i>	70
4.2.2. <i>The Class Firewall of the CUT BankAccount</i>	71
4.2.3. <i>Test Order of the Class Firewall of the CUT BankAccount</i>	73
4.3. <i>JGL Example</i>	78
4.3.1. <i>The ORG Presentation of JGL Sample</i>	79
4.3.2. <i>The Class Firewall of JGL Sample for the CUT ForwardIterator and Test Order of the Class Firewall</i>	80
4.3.3. <i>Some Statistical Figures on the Execution Time of JFireClass on JGL</i>	81
 5. Chapter 5 - Polymorphism and Proposed Solutions	82
5.1. <i>Demonstration of Dependencies Missing due to Polymorphism in the Current Class Firewall Model</i>	83
5.2. <i>Dynamic Binding in Java</i>	85
5.3. <i>The Polymorphic Bottom Substitution Solution</i>	86
5.4. <i>Proof of the Polymorphic Bottom Substitution</i>	89
5.5. <i>Revisiting the Bank Example with the Computation of the Class Firewall</i>	91
5.6. <i>The Typed-Constraint List Solution</i>	92
5.7. <i>Discussion on the Difficulty of Detecting Aggregation Relationships Involving Polymorphic Data Structures (Aggregation Inferencing)</i>	93
 6. Chapter 6 - The Implementation of JFireClass	97
6.1. <i>Introduction to JFireClass</i>	97
6.2. <i>Advantages of ByteCode Analysis</i>	99
6.3. <i>The Class File Parser Component</i>	101
6.3.1. <i>Class File Structure</i>	101
6.3.2. <i>Implementation of the Class File Parser</i>	104
6.3.2.1. <i>Construction of the Class Parser</i>	105
6.3.2.2. <i>Parsing Process of Class File Objects</i>	106
6.4. <i>The Lightweight Reflection Component</i>	108
6.4.1. <i>Construction of the Lightweight Reflection Package</i>	109
6.4.2. <i>Class Information Refining Process</i>	111
6.4.2.1. <i>The descriptors Attribute</i>	111
6.4.2.2. <i>Decoding the descriptors</i>	113
6.4.2.3. <i>The access_flags Attribute</i>	114
6.4.2.4. <i>Decoding the access_flags Attribute</i>	114

List of Figures

Figure 1-1	An ORG representation of a simple Car example	3
Figure 1-2	The class firewall of the CUT Builder	4
Figure 1-3	Missing polymorphic dependencies in an ORG	6
Figure 2-1	An example of inner classes	20
Figure 2-2	An example of the transformation of inner classes in a source code at compile time to generate bytecodes	21
Figure 3-1	Class change identification at the class level and the effects	43
Figure 3-2	The class firewall algorithm - a modified version of DFS [Cormen et. al., 90]	49
Figure 3-3	The class firewall of the CUT Builder and Tire	52
Figure 3-4	The outline of the test order algorithm [Kung et.al., 95a]	54
Figure 3-5	The outline of the alternative test order algorithm	58
Figure 3-6	The topological sort algorithm [Cormen et. al., 90]	59
Figure 3-7	The Strongly Connected Components algorithm [Cormen et.al., 90]	59
Figure 3-8	A sample of code demonstrating a newly found type of cycles, the inner class aggregation cycle	62
Figure 3-9	The formation of the inner class aggregation cycle	62
Figure 3-10	The outline of a modified DFS algorithm to break cycles in a cluster	63
Figure 4-1	A full presentation of a class	68
Figure 4-2	A short presentation of a class	68
Figure 4-3	A full presentation of an interface	68
Figure 4-4	A short presentation of an interface	68
Figure 4-5	A Class diagram for the Bank Example	70
Figure 4-6	An ORG representation for the Bank Example constructed from JFireClass	71
Figure 4-7	The class firewall of the BankAccount class	71
Figure 4-8	The application of the first DFS on the Bank's ORG	74
Figure 4-9	Production of clusters from the second DFS on the transpose ORG	75
Figure 4-10	Production of a minor test order from the last DFS	77
Figure 4-11	A test order of the BankAccount firewall generated by JFireClass	77
Figure 4-12	A test order of the entire Bank Example generated by JFireClass	78
Figure 4-13	The ORG representation for the JGL sample	79
Figure 4-14	The class firewall and test order of the CUT ForwardIterator generated by JFireClass	80
Figure 5-1	A sample of the Bank example with dependencies missing due to polymorphism	84
Figure 5-2	The ORG representation of the Bank Example with the polymorphic bottom substitution	88
Figure 5-3	The general ORG accounting for polymorphism	89
Figure 5-4	Class firewalls of the PreferredCustomerCheckingAccount before and after polymorphism handled	91
Figure 5-5	A fragment in class Bank definition	94
Figure 5-6	An example of how the PDS aggregate is used	94
Figure 5-7	An example of a class without "true" dependency with its PDS aggregate	96
Figure 6-1	JFireClass 1-1	97
Figure 6-2	File class structure	101
Figure 6-3	The Method_info structure and its nested attribute structure	102
Figure 6-4	A snapshot of class file class class definition – the JcfClassFile	105
Figure 6-5	A snapshot of the method and field class definition – the JcfMembers	106
Figure 6-6	Class definition of JcfLocalVariablesAttribute mapping out with the predefined LocalVariable attribute of class file structure	106
Figure 6-7	A full implementation of a JcfClassInput's read method to read the class file input stream completely	107
Figure 6-8	The readAttribute method serving as attribute dispatcher – The second method depicting how a particular local variable attribute object reads itself from the input stream	108

Figure 6-9	LWClass, at construction time, loads all needed information for dependencyanalysis	110
Figure 6-10	An example of constructing the LWClass object's <i>methods</i> attribute	110
Figure 6-11	A sample of types encoding according to JVM specification	112
Figure 6-12	The getFullType pseudo-code in LWDscriptor class	113
Figure 6-13	A sample of modifier's values	114
Figure 6-14	A sample of codes to show how values in a method's modifier are retrieved	115
Figure 6-15	A pseudo-code to return all types of relationships of a class	116
Figure 6-16	An algorithm to construct the ORG object from the given set of LWClass objects	117
Figure 6-17	The implementation of the bottom forwarding solution to polymorphism	118
Figure 6-18	The pseudo algorithm to compute leaves for a polymorphic class node	119
Figure 6-19	A sample of a type-constraint list for the Bank class	120
Figure 6-20	The JFireClass's GUI components	121
Figure 6-21	The Custom FileDialog's aiding in selecting input process	122
Figure 6-22	Transforming class files into LWClass objects	123
Figure 6-23	Displaying the ORG of the Car example	124
Figure 6-24	Displaying the ORG of the Car example in the UML short form	124
Figure 6-25	The class firewall for the single Irun node	125
Figure 6-26	The class firewall for the set {Irun, Engine}	126
Figure 6-27	A test order for the CUTs {Irun, Engine}	127
Figure 6-28	A test order for the entire Bank Example	127
Figure 6-29	The typed-constraint list for the Portfolio class in the Bank example	128
Figure 6-30	Updating the ORG accordingly to the typed-constraint list of the Portfolios class	130

Chapter 1 - Introduction

1.1 Introduction

The Object-Oriented (OO) paradigm has been gaining wide acceptance as an implementation technique in the software industry for the past few years [Kung et. al, 95b], [Lejter, 92]. Incorporating features such as polymorphism and inheritance, the OO paradigm offers a great level of software reuse and flexibility [Booch, 86], [Wilde&Huitt, 92]. However, these features complicate software testing and maintenance [Wilde&Huitt, 92], [Turner&Robson, 93]. As a result, they give rise to the demand for additional techniques to aid in these activities. To date, much of the current literature on OO systems focuses on the early phase of the system development life cycle [Coad&Yourdon, 91], [Jacobson et. al., 92]. Little time has been spent on object-oriented testing methodologies, particularly in regression testing.

Regression testing is applied to modified software to provide confidence that the system still functions according to its specifications. Regression testing is an expensive software maintenance activity [Moreton, 90], [Rothermel&Harrold, 94a]. Ideally, regression testing should be applied to "those parts that are affected by the modification to save effort and time" [Kung et. al., 95a]. The task of identifying these affected parts is called *software change impact analysis*, or *impact analysis*, for short. Impact analysis consists of identifying the consequences of a change that is introduced to a software system. [Bohner&Arnold, 96] stress impact analysis is "the act of identifying the potential effects of a change, or estimating what needs to be modified to accomplish a

change”. According to [Bohner, 91], impact analysis tends to focus on changes at source code level: “Usually, only system source code is analyzed to reflect the ripple effect after a change has been made”. A ripple effect is defined as the “effect caused by making a small change to a system, which affects many other parts of a system” [Stevens et. al., 74]. Dependency (impact) analysis, which focuses on the dependency relationships that exist among program entities (i.e., data, modules), has been receiving a great deal of attention [Moser, 90], [Horwitz et. al., 90], [Rothermel&Harrold, 94a]. However, these analysis techniques are not adapted to OO systems. One of the main reasons is that none of the techniques addresses the various relationships that exist between classes, such as inheritance, aggregation, and association [Kung et. al., 95a].

[Kung et. al., 95] propose a new approach, called “class firewall”, adapting the concept of firewall from [Lueng&White, 92], to perform dependency analysis on C++ systems at the class level. A class firewall identifies all of the classes affected by changes made to a class. The source code is modeled by an *Object Relation Graph* (ORG), which is comprised of a set of nodes (i.e., classes) and a set of edges (i.e., dependency relationships). The dependency relationships can be categorized into the inheritance, aggregation, and association relationship. The following figure illustrates an ORG representation of a simple Car example.

Figure 1-1 An ORG representation of a simple Car example

Nodes represent classes and three-letter-word labeled arrows represent dependency edges. For example, the *Inh*-edge (Car, Vehicle) means the Car node's class depends on the Vehicle node's class via the *inheritance* relationship. Similarly, the *Agg*-edges and *Ass*-edges indicate the *aggregation* relationships and *association* relationships respectively.

The class firewall of the class under test (CUT) can be determined by computing the transitive closure on dependency relationships on the ORG, starting at the CUT node. The transitive dependency relation simply means that if class A depends on class B, and class B depends on class C, we can deduce that class A depends on class C. The class firewall identifies all classes related to a CUT that may have to be re-tested. Thus the regression testing of the CUT becomes simpler.

Let us compute the class firewall for the CUT Builder. From figure 1-1 above, we deduce that the Vehicle class depends on the Builder class via the association relationship, and the Car and Motorcycle classes depend on Vehicle under the inheritance relationships. Similarly, the class Driver depends on the Vehicle class under the association relationship. By transitivity on dependency relations, the Car, Motorcycle, and Driver classes depend on the Builder class. Thus, a change in the Builder class would affect the Vehicle, Car, Motorcycle and Driver. In other words, the class firewall of the CUT Builder must enclose the Vehicle, Car, Motorcycle, and Driver (and the CUT itself). Figure 1-2 below depicts the class firewall of the CUT Builder.

Figure 1-2 The class firewall of the CUT Builder

But in what order should those classes be tested so as to further reduce the test effort? The class firewall theory also extends itself to generate a test order that can serve as a guideline for re-testing classes at class unit level and class integration level [Kung et.

al., 95a]. The idea comes from the observation that independent classes should be tested prior to dependent ones so that the information of test stubs and test drivers of independent classes can be reused on their associated dependent components. In figure 1.2, we deduce that the class Builder is independent so it should be tested first. Once the class Builder is tested, the Vehicle can be tested immediately because it only depends on the tested Builder class. The Motorcycle, Car, and Driver classes can be independently tested right after the completion of the testing of the Driver class. Thus, one instance of the test order is {Builder, Vehicle, Car, Motorcycle, and Driver}. The underlying algorithm of the class firewall and test order is based on the transitive closure algorithm of Warshall in [Aho et. al., 83] which is considered time-consuming since it executes in $O(n^3)$, where n is the number of nodes (i.e., classes) [Bohner&Arnold, 96].

Moreover, the above approach to OO impact analysis includes some major shortcomings, notably its lack of recognition of polymorphism. The difficulty lies in the fact that polymorphism is a runtime characteristic and, therefore, dependencies triggered by polymorphism are very difficult to be detected by compile-time dependency analysis. [Wilde&Huitt, 92] are quite clear on this characteristic: “a given variable may refer to an object of any class so, using polymorphism, the method to be executed when a given message is sent to it will be determined at run time by the class of the object it represents at that instant”. This implies that there are some forms of polymorphic dependency left out of the current class firewall model.

Returning to the Car example above, figure 1-1 shows that the Driver class depends only on the Vehicle, not on Vehicle's subclasses (i.e., Car and Motorcycle). However, Driver cannot use the Vehicle class directly; instead the Driver must use either Car or Motorcycle for transportation in reality. That means the Driver class also depends on the Car and Motorcycle classes. From the ORG standpoint, it is desirable to have additional edges that show the dependencies of the class Driver on the Car and Motorcycle classes as in the following figure (1-3):

Figure 1-3 Missing polymorphic dependencies in an ORG

The dotted edges from the Driver class node to the Car and Motorcycle class's nodes indicate the dependencies missing due to polymorphism. With these additional edges, we can ensure that the class firewall of Car (or Motorcycle) would enclose the Driver class.

1.2 Motivations

The main motivations behind this thesis arise from three factors:

First, few of the existing regression testing methodologies are well suited for OO systems. They are mostly designed for non-OO languages [Kung et. al., 95a], [Lejter et. al, 92].

Second, there is very little in terms of literature that addresses the issue of software maintenance on OO systems, and especially on regression testing [Rothermel&Harrold, 94a]. There are few automated testing tools, especially for OO [Sharon, 96]. Of those existing tools, few address the effect of OO on regression testing.

Finally, dependency analysis entirely depends on source code structure/syntax to collect impact information. Since impact information is collected from the source code and since every language is structurally and syntactically different from every other, the amount of impact information collected will vary from one language to the next. Because the Java language is relatively new, there is very little in the literature addressing dependency analysis involving it. Thus, it deserves a thorough examination in order to bring regression testing into its folds.

1.3 General Contributions

It is our claim that there are four major contributions of this thesis:

The first contribution is to identify class and object relationships in Java. Along with the three known relationships, i.e. inheritance, aggregation, and association as described in [Kung et. al., 95a], we also detect the interface relationship and the polymorphic inheritance, polymorphic aggregation, and polymorphic association relationships [see Chapter 3]. We also provide proofs, wherever appropriate, on how these new relationships affect regression testing.

The second contribution is to simplify the algorithmic strategy for finding test order as presented by [Kung et. al., 95a]. Our revised version significantly reduces the complexity of converting a cyclic graph to an acyclic one from $O(n^3)$ to $O(n^2)$. In addition, we use the same underlying algorithm to compute both the class firewall and the test order so that the overall approach is simpler. Furthermore, we also take into account the couplings that exist between classes (i.e., the interface relationship). This is a necessary step to ensure the feasibility of the test order algorithm. In addition, we describe our view on the class firewall computation in practice.

Third, we also propose two simple techniques to capture the dependencies due to polymorphism. Taking advantages of Java language constructs (semantics/syntax), we are able to identify certain forms of polymorphic dependency relationships.

The final contribution is our proposal of a tool, JFireClass, that implements the concepts presented in this paper. Basically, JFireClass is capable of constructing the ORG from the given Java software system, of computing the class firewall for a particular class or a set of classes, and of producing the test order of the computed class firewall or of the entire input. JFireClass also supports the UML [Booch et. al., 98] as an alternative mechanism to display the ORG. This offers a convenient way for software developers to compare the design model, which is likely modeled in UML, with the reconciled implementation model constructed from the tool.

1.4 Outline of the Thesis

The remaining chapters of this thesis are organized as follows:

In chapter 2, we review some OO concepts. This chapter is not intended as a comprehensive review of the technology, but rather as an overview of those OO concepts that are of direct relevance to the thesis. In section one, we discuss OO concepts in general. The focal point of this section is the review of object and classes notions along with their relationships. In section two, we examine Java. In the last section, we cover background information on OO testing. Regression testing and dependency analysis are also visited.

Chapter 3 introduces the concept of class firewall. It emphasizes the identification of Java class relationships, the construction of the underlying graphical representation of

the class firewall (i.e., ORG), and the algorithm to compute a class firewall (i.e., transitive closure). Chapter 3 also provides the algorithm to generate a test order. A major section is dedicated to describing our algorithm of the test order including a proposed version of cycle breaking algorithm, which will be proved to be simpler and more efficient than those described in [Kung et. al., 95a].

Chapter 4 presents two examples for constructing the ORG and generating a test order. A concrete example that uses the JGL package, a commercial software product, illustrates the potential of the tool.

In Chapter 5, we review the polymorphism issue and then describe simple solutions, called *polymorphic bottom substitution* and *typed-constraint list*, to capture polymorphic dependencies. Aggregation inferencing [Hurdle, 97] is also discussed.

Chapter 6 offers a detailed discussion on the implementation of JFireClass. Unlike traditional approaches, the impact analysis is not carried out on the system source code but on the system Java bytecodes. Section 6.1 describes the structure of the Java bytecodes and the reasons why we use bytecode instead of source code.

The final chapter concludes the work and proposes future research topics in the area of OO regression testing.

Chapter 2 - Background

This chapter provides background information required in understanding class firewall theory in Java. A section highlighting those Java features that are of interest to this paper will follow a brief overview of OO concepts. The last section will then review OO testing methodologies.

2.1 Object-Oriented Concepts

[Jacobson et. al., 92] defines **object-orientation** as a technique for system modeling in which a system can be described as a number of objects that interact. Object-oriented programming (OOP) is a programming approach based on this concept. OOP encompasses the principles of *abstraction*, *encapsulation*, *inheritance*, and *polymorphism*. [Booch, 94] points out:

OOP is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

An **object** is an encapsulation of behavior and state. An object implements its behavior using methods and maintains its state through its instance variables. For example, a car is accelerating and is in a *running* state when its speed is 70 km/h, and its transmission is in its third gear. Objects in the real world interact with each other to perform tasks. Objects in OO world interact in the same way by calling each other methods. The

blueprint for an object is called a class. More precisely, a **class** defines common data and behavior shared by different objects or instances of the same class. In addition, objects and classes enrich the concept of abstract data type through an inheritance capability.

Inheritance is the process by which classes share or reuse similar features. A parent class, called the (direct) superclass, usually is a generalization of a child class or subclass. A subclass is a specialization of a super class. Classes can be organized into a hierarchical tree. For example, all Java and Smalltalk classes are rooted to a single class called *Object*. Classes at lower levels in the tree (i.e., descendant classes) can inherit the state and behavior of classes at higher levels in the tree (i.e., ancestor classes). State and behavior of the Object class are accessed and used by all other classes. Single inheritance refers to a class that has only one superclass, as opposed to multiple superclasses. The latter defines multiple inheritance as available in C++.

An important OO principle is polymorphism. **Polymorphism** reflects the ability that different objects can respond to the same message but behave in different ways.

[Jacobson et. al., 92] defines polymorphism as follows:

Polymorphism means that the sender of a stimulus does not need to know the receiving instance's class. The receiving instance can belong to an arbitrary class.

For example, class Car inherits the state and behavior from the class Vehicle by being a subclass. The example can be extended by creating a class Motorcycle as a subclass of

Vehicle. Now both the Motorcycle and Car classes understand the same set of messages (method names) that are defined in their shared superclass, for example the message *drive*. However, they may behave differently (i.e., some methods may be rewritten from the inherited classes – a motorcycle drives differently from a car). We can also incorporate a Person class into the model. A person that is going to rent a vehicle may not need to know exactly *what kind of vehicle*, but does need to know if he can “drive” the rented vehicle or not. Without this knowledge, the person would have to specify what kind of vehicle. This forces the renter to know which types of vehicle are available in advance. Therefore, our Vehicle class could simply include a “drive” method that would be inherited by both Motorcycle and Car subclasses. Polymorphism is useful since the sender usually does not need to know the actual class of the receiver until run time. The only thing the sender cares is that the actual receiving instance at run time must comply with the established interface of the method.

Unfortunately, polymorphism also has a negative impact on software maintenance and testing. [Wilde&Huitt, 92] stresses that polymorphism greatly complicates the tracing of dependencies because the actual object that carries out the operation is not known until runtime; [Corriveau, 96] also points out that polymorphism reduces testability.

The following section is devoted to identifying relationships among objects and classes in OOP, especially in Java.

2.2. Classes and Objects Relationships

A *relationship*, according to [Booch et. al., 98], is a semantic connection among elements (i.e., classes and objects). There are many ways that OO “elements” are connected.

2.2.1 Inheritance Relationships

An *inheritance relationship* is a relationship between a class (subclass) and its superclass. The inheritance relationship can be further categorized into three types based on its semantic intent [Salus, 98]. The first form is the *strict-subtype relationship* where a subclass leaves the inherited methods and attributes intact but may add new methods and/or attributes. An example is a Customer that extends the Person class by having additional behavior specific to a customer yet without modifying any inherited method or attribute. The second form is the *subtype or polymorphic inheritance relationship* where a subclass (1) must implement certain methods that are declared in its superclass, (2) and/or could override inherited methods, (3) and/or could add new methods. An example is a Car and a Motorcycle that both extend from Vehicle by implementing and modifying certain behavior of Vehicle. In Java, a superclass that involves in such a polymorphic inheritance relationship is usually declared with the *abstract* keyword along with some *abstract* methods whose implementations are provided by each of its concrete (or non-abstract)¹ subclasses or descendants. The third form is the *subclassing relationship* where the subclass reuses the inherited implementation but defines a completely new abstraction (i.e., external interface or

¹ A non-abstract class is a class that is not declared with the *abstract* keyword.

public methods) that is totally different from that of the superclass. For example, a class named `List`, which is inherited from the `Queue` class, is defined with a new set of external interface (related to `List`) whose implementations are basically reused from the superclass `Queue`. This type of inheritance is sometimes referred to as *inheritance for implementation*.

2.2.2 Interface Relationships

An *interface relationship* is a relationship between a class and an interface that the class “inherits”. In Java, an interface is a special type of abstract class, a pure abstract class, whose attributes must be constant and behaviors are specified without its implementation. When a class implements an interface, it ensures clients of the class that the methods declared in the interface are implemented. The interface relationship is polymorphic since it guarantees that different objects of the same interface provide the implementation for the same set of methods. For instance, the `Vehicle` class that implements the `Drive` interface must provide the definition of the single *drive* method that is declared in the `Drive` interface, so that any objects that use the `Vehicle` class or its descendants are guaranteed that the object supports the *drive* method. Unlike other OO languages, Java offers syntax to implement the notion of an interface and that helps to address the traceability and testability issues related to polymorphism.

Objects can collaborate with others in two ways: whole-part relationship and association relationships.

2.2.3 Aggregation (Whole-Part) Relationships

A composite object can be comprised of several different components (i.e., other objects). The relationship between a composite object and its components is called a *whole-part relationship*. The composite object is referred to as the whole, whereas its components are referred to as the parts. For instance, a car (i.e., the whole) has an engine, a body, and tires as its parts. Parts that live and die with the whole are referred to as being in *composition relationship*, otherwise we have an *aggregation relationship*². For example, there is a composition relationship between a tire and its valve. Once, the tire is destroyed, the valve is destroyed along with it. On the other hand, when a windshield is replaced, its wipers can be reused. We say the windshield and its wipers are in the aggregation relationship.

2.2.4 Association (Use) Relationships

An *association relationship* or *use-relationship* indicates that two-independent objects associate with each other in some manner. This type of relationship can be found in data dependencies, message passing, and parameter passing [Kung et. al., 95a]. Going back to the car example, the car cannot operate without *gasoline*. The car uses *gasoline*, which is neither part of a car nor a generalization and an interface of the car. Thus, the car relates with the *gasoline* via the association relationship.

² This paper groups both of these relationships into *aggregation* relationships.

2.3 Java as an OO Language

As was mentioned earlier, JFireClass aims at supporting impact analysis of Java program. In order to develop this idea, this section first summarizes the relevant concepts of the language. In addition, we will focus on the mechanism that allows classes to be enclosed in other classes: inner classes. Understanding how inner classes collaborate with their enclosing class and with other outside classes allows us to categorize their relationships properly.

2.3.1 Java in General

As with any OO language, objects in Java are instantiated from a class. The class *Object* is the root of all Java classes. The Java syntax for a class definition is similar to that of C++. However, generic or template classes are not supported. Java supports single inheritance: a class has one and only one direct superclass. However, it can implement multiple interfaces. In Java, both static class variables and static class methods are supported. They are associated with the class itself rather than with the instance of the class. Polymorphism in Java is enforced by interfaces. Java supports nested classes, best known as *inner classes* [described in section 2.3.2].

Java is a strongly typed language. The type of objects is statically checked at compile time. Conversely, Java supports late binding meaning that methods generally bind to their actual implementation at runtime. Interestingly enough, Java also supports the performance benefits of early binding: “[w]hen a compiler can determine that a method cannot be overridden by subclassing, the method definition [i.e., implementation] is

bound to the method class at compile time.” [Grant, 97]. This mechanism is made possible by the keyword *final* (and *static*). If a class or a method of a class is declared with *final* keyword, it means that the class cannot be subclassed or the method cannot be overridden. Thus, the Java compiler knows exactly the actual method to be called at runtime. Dynamic loading is also supported. This feature refers to the fact that a class is not linked at compile time but loaded as needed at runtime.

Java is both a compiled and interpreted language: Java source code is compiled to Java bytecodes, which is then interpreted by a Java Virtual Machine (JVM), Java’s runtime environment. The specifications of the Java language and JVM are fully defined; there are no implementation-dependent details. This makes Java an extremely portable language [Grant, 97].

2.3.2 Inner Classes

Inner classes are classes defined inside of other (enclosing) classes. The enclosing classes can have as many inner classes as they require. Inner classes cannot be used outside the scope of the enclosing class. This type of restriction raises an interesting point of dependency. On one hand, it increases the coupling between the enclosing class and its inner class by allowing the inner class accessing certain properties of the enclosing class. On the other hand, it decreases the coupling between one subsystem (package³) to the other by not allowing classes of one subsystem accessing inner classes

³ Package is mechanism to organize logically related classes into groups. A Java system is normally composed of several packages, each of them may contain sub-packages and so on.

of the other. Understanding inner classes is an essential step to properly establish the relationships between inner classes and their enclosing class.

Inner classes can be categorized according to the scope of their declaration. The following excerpts offer definitions of the four types of inner classes and can be found in [Flanagan, 97].

1. A ***nested top-level class or interface*** is defined as a static⁴ member of an enclosing top-level class or interface. The definition of a nested top-level class uses the static modifier, just as the definition of a static method or static field does. Nested interfaces are implicitly static and so are always top-level. A nested top-level class or interface behaves just like a "normal" class or interface that is a member of a system.
2. A ***member class*** is also defined as a member of an enclosing class, but unlike a nested top-level class, it is not defined with the static modifier. In many ways, a member class is analogous to the other members, the instance fields and methods of a class. Every instance of a member class is associated with an enclosing instance of the class that defines it.
3. A ***local class*** is an inner class defined within a block of Java code; it is visible only within that code. Interfaces cannot be defined locally. Because a local class is defined within a block of code, it is analogous, in some ways, to a local variable. Local classes are not member classes, but can still use the instance variables, class variables, and methods of enclosing classes. More importantly, however, the code

⁴ Declared with *static* keyword

within a local class definition can use any final local variables or parameters that are accessible in the scope of the block that defines the class.

4. An **anonymous class** is an extension to the local class concept described above. Instead of declaring a local class with one Java statement, and then instantiating and using it in another statement, an anonymous class combines the two steps in a single Java expression. An anonymous class, as you might guess, does not have a name. And because it is instantiated in the same expression that defines it, it can only be instantiated once. Except for these differences, anonymous classes are quite similar to local classes in behavior and use.

The following example illustrates different types of inner classes:

```

Class EnclosingClass {
    static class myNestedTopLevelClass { //class definition};           // top-level class
    public class MyMemberClass { //class definition};                   // member class (non-static)
    void foo1 { ....
        class myLocalClass { // class definition here };               // local class
        ...
    }
    MyAnonymousLocal foo2 { ...
        return new MyAnonymousLocal{ // class definition here }; // anonymous local class
    }
}

```

Figure 2-1 An example of inner classes

One important aspect of the inner class feature is that it is entirely implemented by the compilers and that no changes to the JVM are required to support them. Basically, a Java compiler transforms an inner class in such a way that converts the inner class to a regular top-level class. In addition, to ensure the availability of the reference of the

enclosing class for the inner class, the compiler automatically adds an extra private⁵ instance variable in the member class to hold the reference of the enclosing class. This instance variable is initialized from an extra argument passed to the inner class constructor.

Consider the following example:

Source Code	Transformed Source Code on the fly
<pre> class EnclosingClass { int a = 10; InnerClass getInnerClass() { return new InnerClass(); } class InnerClass { int b; public InnerClass(){ b = a; } } } </pre>	<pre> class EnclosingClass { int a = 10; InnerClass getInnerClass() { return new InnerClass(this); } } class EnclosingClass\$InnerClass { EnclosingClass this\$0; int b; public InnerClass(EnclosingClass this\$0){ this.this\$0 = this\$0; this.b = this\$0.a; } } } </pre>

Figure 2-2 An example of the transformation of inner classes in a source code at compile time to generate bytecodes

In the above example, the inner class *InnerClass* is transformed to a regular class *EnclosingClass\$InnerClass*, and the inner class receives the instance of the enclosing class as one of its attributes. The fact that an inner class implicitly contains the instance variable of this enclosing class allows us to properly identify relationships between them that would normally not be seen in the source code.

Inner classes will be revisited in Chapter 3 with respect to class relationship identification and test order algorithm, and in Chapter 6 with respect to the implementation of *JFireClass*.

⁵ Declared with *private* keyword

2.4 OO Regression Testing and Software Change Impact Analysis

Software testing is the process of examining software with the intent of finding errors. [Meyers, 79] says, “A test that reveals a problem is a success. A test that did not reveal a problem was a waste of time”. The ultimate goal of software testing is not only to find errors but also to correct these errors. Thus, systematic techniques that generate test cases to provide coverage of all possible input are required. There are three major categories of testing activities: testing the development process and the supporting documents, testing the analysis and design models, and testing the implementation. The first two activities are beyond the scope of this paper; interested readers are referred to [McGregor&Korson, 94]. The objective of implementation testing is to locate errors in the final source code. Static testing refers to any testing that can be done without actually executing software. Typical examples of these techniques are structured walkthroughs and code inspections [Yourdon, 78]. Dynamic testing consists of any testing that involves running the actual software to find errors in the code.

The OO paradigm comprising such features as inheritance, encapsulation, polymorphism, and dynamic binding, complicates the testing process [Wilde&Huitt, 92], [Lejter et.al., 92]. [Perry&Kaiser, 90] stresses that the productivity that is gained from the OO development process diminishes partially during testing. In addition, OO requires different approaches to testing as compared to conventional testing methodologies [Firesmith, 93], [Perry&Kaiser, 90]. Most of the conventional approaches focus on either structural or behavioral characteristics of the software. These

are insufficient because OO systems simultaneously embody both of those characteristics.

2.4.1 Regression Testing

Regression testing (RT) is done during software maintenance with the purpose of re-establishing the confidence that the software will continue to function after some modifications. The goal of regression testing is to ensure that no new errors into existing functionality are introduced and no unintended side effects are caused by the changes. Regression testing can be categorized into two types based on the possible of specification and/or code changes [White, 91]:

- (1) *progressive regression testing* that involves a modified specification or user requirement. Whenever new enhancements, new data or functions are incorporated in a system, the specification is modified to reflect these changes. In most cases, new modules, subsystems will be added to the system with the consequence that RT process involves testing a modified program against a modified specification.
- (2) *corrective regression testing* that involves solely code changes. Only some instructions of the program and possibly some design decisions are modified. This has important implications because most of the test cases in the previous plan are likely to be valid.

Any progressive regression testing strategy could naturally include a corrective regression testing strategy. Because progressive regression testing deals with changes in the specifications, not only should old test cases be reused for testing but new test cases should also be generated to satisfy some adequacy criteria. With respect to reusing old

test cases, there are two alternatives - either running all existing tests again, or only re-running that subset of tests that exercises the changes. The later is referred to as *selective regression testing* and, because it requires less test effort, is more cost-effective. Software methodologies stress the necessity of this strategy due to the high cost of software maintenance. In particular, regression testing is an expensive software maintenance activity [Moreton, 90], [Rothermel&Harrold, 94a]. [Rothermel&Harrold, 94a] are quite clear on this point:

[T]he cost of software maintenance dominates the overall cost of software... Because regression testing constitutes a significant percentage of software maintenance, improvements in regression testing processes can significantly lower the overall cost of software.

According to [Kung et. al., 95a], a selective regression testing strategy should involve four essential activities:

- Identifying the affected components and characterizing the ripple effects of the modification,
- Finding a cost-effective and optimal class testing sequence, called the *test order*. Ideally, we want to minimize the cost and test effort.
- Selecting the test cases to retest the affected components based on the generated test order. This involves reusing the existing test cases, modifying them if needed, dropping obsolete test cases, and generating new ones.

- Retesting the modified program with a regression test plan. Testers must ensure that the modified program still performs the intended behaviors specified in the specifications.

The first activity is the heart of the selective regression testing strategy, best known as *software change impact analysis*, or *impact analysis* for short [described in the following section 2.5.2]. This paper only focuses on the first two activities.

2.4.2 Software Change Impact Analysis

Software change impact analysis (IA), according to [Bohner&Amorl, 96], is defined as “identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change”. In particular, IA allows testers to understand the impact of changes by identifying affected components. Over the years, a considerable number of IA techniques have been proposed. [Bohner&Amorl, 96] lists typical techniques as follows:

- Using cross reference listings to see what other parts of a program contain references to a given variable or procedure
- Using program slicing [described later] to determine the program’s subset that can affect the value of a given variable
- Browsing a program by opening and closing related files
- Using traceability relationships [described later] to identify changing artifacts

- Using configuration management systems (CMS) to track and find changes. [The CMS system is often capable of check-in /check-out source code, displaying differences of two source-code versions, displaying the calling structure of a program, etc.] and
- Consulting designs and specifications [or using language specific tools] to determine the scope of a change.

These techniques have been used in two major approaches of IA: traceability analysis and dependency analysis. ***Traceability analysis*** tracks the relationships among software artifacts of the development lifecycle (design, implementation, etc.), whereas ***dependency (source code) analysis*** tracks the relationships within source code entities (variables, modules, etc.). [Bohner&Arnold, 96] stresses that “these complementary areas approach IA from quite different perspectives and have their respective advantages [in] enhancing the potential of identifying software impacts”.

Most dependency analysis techniques are basically comprised of a graph presenting a dependency model such as a *system dependence graph* or a *control flow graph*; and an underlying algorithm that gathers the impact information on the supporting representation, such as transitive closure, inferencing, and program slicing [Bohner&Arnold, 96]. The definition of those algorithms can be found in [Bohner&Arnold, 96]:

- Transitive closure is a fundamental IA technique. Transitive closure algorithms take a graph of objects and relationships as input and determine all objects reachable from any object.

- Inferencing systems use rules to characterize relationships among objects. The rules define the search space or how it is explored, as well as the underlying relationships among entities.
- Program slicing provides a mechanism for constraining the view and behavior of a program to a specific area of interest. A program slice is an executable program that includes only the parts of the original program that is affected by the change. The slice is taken from data and control flow information analyzed with respect to some slicing criteria. IA that uses such a program slicing approach requires the target program to be executed as part of detecting impacts and is referred to as *dynamic impact analysis*. Other approaches are known as *static impact analysis* strategies (i.e., class firewall).

Dependencies that occur within a program can be further categorized into two major categories: data dependencies and control dependencies. The following definitions are found in [Bohner&Arnold, 96]:

- Data dependencies are relationships among program statements that define or use data. A data dependency exists when a statement provides a value used directly or indirectly by another statement in a program. Data definition/use graphs are typical representations of these dependencies. Data-flow analysis produces dependency information on what data goes where in the software system. [For example, if module A defines a variable, say g, which is then used by module B, then module B is said to be data dependent on module A. Modules A and B is referred to as a definition/use pair].
- Control dependencies are relationships among program statements that control program execution...Program dependency graphs are often used to account for control dependency. Control-flow analysis provides information on the logical

decision points in the software system and the complexity of their structure. Control-flow techniques identify procedure-calling dependency, logical decisions (conditions such as IF-THEN-ELSE, LOOPS, CASE statements), and under what conditions these hold true.... [For example, if module A calls module B then module A is referred to as procedure-calling dependency].

2.4.2.1 Past Work on Dependency Analysis

[Harrold&Soffa, 89] introduces a data dependency analysis technique to identify the change effects within a module. The idea is to use a control-flow graph to identify the affected definition-use pairs. This technique is also extended to evaluate the effect at the inter-procedural level, which accounts for the calling context. However, relying only on data dependency, this technique may miss other effects that may be exposed by other forms of dependencies [Rothermel&Harrold, 94b]. Similarly, [Moser, 90] also proposes an impact analysis approach to trace data dependency information within an Ada program. The technique is called a compositional method that combines the data dependency graph and specific Ada composition rules (iteration, recursion, exception handling, and tasking).

[Horwitz et. al., 90] proposes a slicing technique to identify the change effects between modules. The authors proposed a system dependency graph to identify the affected control paths and definition-use pairs based on their proposed inter-procedural slicing technique where the slice cross the boundaries of procedure calls to account for transitive dependencies involved in procedure calls within a program.

[Lueng&White, 92] initially proposes the concept of *firewall*, a control-flow dependency analysis for regression testing, to enclose all affected modules, which must be retested, due to changes introduced in one or more modules. The control-flow dependency is constructed from a *call graph*, where nodes are modules and arrows indicate the dependencies of the calling modules to the called modules. Later, the authors extend the notion of firewall to account for the presence of global variables. The basic idea is based on the dependency of modules that define global variables to those that are referenced (definition/use pairs). A data-flow graph is presented to support the construction of this data-flow dependency.

[Harrold et. al., 93] propose a *unified inter-procedural graph* (UIG) that combines all existing program representations such as call graph, system dependency graph, etc. The proposed UIG allows us to extract all information such as data flow, control flow, data dependence, and control dependence to provide an integrated view for maintenance tasks. Algorithms are also proposed to accommodate the construction of the UIG.

[Harrold&Larsen, 96], [Harrold&Liang, 98] propose a concept of slicing techniques for object oriented programs – the *object slicing*, which is based on the system dependency graph (SDG). The SDG has knowledge of classes (base, derived), group of interacting classes, and complete OO systems. This information helps (1) to build the SDG in incremental fashion by reusing class representation, (2) to distinguish data members for different objects, and, (3) to represent the effect of polymorphism on parameter and parameter bindings, etc. An algorithm to implement the SDG is also presented.

[Kung et. al., 95a] refine the concepts of firewall to better fit the OO environment, and propose a strategy for detecting the ripple effect [see Chapter 1] of a change introduced to a particular class in a C++ program. The firewall is built upon a combination of data flow and control flow representations, which the authors defined as an *Object Relation Graph* (ORG). An **ORG** is a set of nodes (i.e., classes) and a set of edges representing the various forms of class dependencies: inheritance, aggregation and association. The ripple effect is computed by determining the transitive dependency of only those relationships originating from the class to which changes have been made, defined as the **Class Under Test** (CUT). Regression testing of the CUT becomes simpler because only those classes related (through transitive dependency) to the CUT have to be retested. The firewall model also generates a cost-effective test order that can serve as guideline for retesting classes at the class unit level and class integration level [Kung et. al., 95a,b]. This idea comes from the observation that independent components should be tested prior to dependent ones so that the information provided by test stubs and test drivers of independent classes can be reused on their associated dependent components. However, the class firewall does not account for polymorphism as defined in here. Consequently, certain affected components may be left out of the firewall although data and control dependencies are taken into account (see figure 1-3 in Chapter 1).

Most recently, [Hurdle, 97] proposed a tool to construct the class firewall in Smalltalk based on Kung's theory. His tool has a few limitations. For instance, it mostly fails to

detect aggregation relationships and does not detect association relationships at all. The chief reason, as Hurdle explained, is due to Smalltalk's dynamic typing.

JFireClass, a reverse-engineering tool for Java programs, will exploit the class firewall theory to provide change impact analysis for Java programs. Given a CUT, *JFireClass* computes the ripple effect of the changes to the CUT by applying a simple Depth-First Search algorithm (DFS) on the corresponding ORG of a given Java code. The dependencies that *JFireClass* take into account include inheritance, association, aggregation, interface, and their refinements⁶ such as polymorphic class inheritance. *JFireClass* also computes the test-order for a given firewall and/or the entire program. The underlying transitive closure algorithm, initially taken from [Kung et. al., 95a], is revised in order to improve its efficiency and simplicity. *JFireClass* is the first dependency analysis tool to tackle the ever-present polymorphism in Java.

The next four chapters will focus on the fundamental building blocks of the tool and its implementation.

⁶ a refinement means a relationship that represents a fuller specification of something that has already been specified at a certain level of details [Booch et.al., 98].

Chapter 3 - Class Firewall in Java

The class firewall is intended to enclose the set of classes that are directly or indirectly affected by changes made to a given class. These classes are related to the changed class by relationships that are extracted from the system. Determining the class firewall involves constructing the corresponding ORG of the target system and applying transitive closure to the dependency relationships. The class firewall encloses *only* affected classes (i.e., most likely a subset of classes in a system) requiring re-integration and regression testing [Lueng&White, 92]. Thus, the test effort is reduced. [Kung et. al., 95a&b] propose a cost-effective test order to further reduce the test effort by dictating the order of testing for individual classes. In this chapter, we will touch upon (1) identifying class relationships in Java, (2) constructing an ORG based on those relationships, (3) defining the class firewall in Java, and (4) presenting the original test order from [Kung et. al., 95a] and our alternative algorithm and discussing the merits of this new algorithm.

The reference material for constructing the ORG and the class firewall can be found in [Kung et. al., 95a&b]. We expand on this work and modify it to reflect new relationships found in Java only, explicitly stating what changes are made vis-à-vis the original work from [Kung et. al., 95a].

3.1 Identifying Class Relationships in Java

This section will focus on identifying the various class relationships found in Java. The class relationships basically include the inheritance, association, and aggregation relationships as described in [Kung et. al., 95a] with some modifications to accommodate Java's language variation, as well as a new relationship - the interface relationship. The addition and modifications are summarized as follows:

1. The inheritance relationship supports the notions of class inheritance and interface inheritance. The class inheritance relationship is also refined to capture the polymorphic property.
2. The aggregation relationship excludes the notion of pointers since Java does not support them. Hidden aggregation relationships between a class and its inner classes (i.e., member, local, and anonymous classes) are recognized.
3. The association relationship covers all refinements described in [Kung et. al., 95a]. In addition, we attempt to provide a comprehensive list of all types of variables involving in this relationship with its containing class.
4. The interface relationship is introduced in this context. The interface relationship is further refined into *strict-data*, in which the interface contains only constant values. The refinement helps us to reduce the numbers of dependencies triggered by polymorphism [see Chapter 5].

It should be noted that for a better understanding of class relationships, we describe the relationships through Java source codes, keeping in mind that the actual implementation is done through bytecode analysis [see Chapter 6].

3.1.1 Inheritance Relationships

3.1.1.1 Class Inheritance Relationships

The inheritance relationship can be detected through Java's class declaration. The *extends* keyword in class declaration specifies the sole superclass of the class being declared. If a class is declared without an *extends* keyword, the class Object is its implicit superclass.

<code>public class Car <i>extends</i> Vehicle</code>	- The dependency of class Car toward class Vehicle is identified as a class inheritance relationship.
<code>private class List</code>	- Class List automatically inherits Object. The dependency of class List toward class Object is identified as class inheritance relationship

3.1.1.2 Polymorphic Class Inheritance Relationship

When a superclass is specifically declared with the *abstract* keyword, the class inheritance relationship becomes *polymorphic* (i.e., we have a polymorphic class inheritance relationship). More details are provided in Chapter 5.

<code>abstract class Vehicle</code>	- The dependency of class Car toward class Vehicle becomes a polymorphic class inheritance relationship.
--	--

3.1.1.3 Interface Inheritance Relationships

The *interface* inheritance relationship occurs between an interface and its superinterface(s) for which the interface extends. Similar to the class inheritance relationship, the *interface* inheritance relationship can be detected through Java's interface declaration. The *extends* keyword in the *interface* declaration indicates the superinterface(s) of the interface being declared. It should be noted that there is no implicit superinterface and that an interface may have more than one superinterface.

interface Jumpable *extends* Hopable

- Jumpable establishes an inheritance relationship with Hopable

interface Jumpable *extends* Goable, Swingable

- Jumpable establishes inheritance relationships with Goable and Swingable
-

3.1.2 Aggregation Relationships

In general, attributes (or fields in Java) of a class are the aggregates of that (containing) class. These aggregates can be identified in the field declarations within a class. If the aggregates are either declared as abstract or are interfaces, the aggregation relationships become *polymorphic* (i.e., we have a polymorphic aggregation relationship). This also applies to all refinements below.

3.1.2.1 Static Aggregations

Aggregates that are declared with the *static* keyword are referred to as static aggregates. Hence, the containing class has static aggregation relationships with its static aggregates. These aggregates are often referred to as *class variables*.

```
class Car {
    static Speed theSpeed;    // The aggregate theSpeed of type Speed is declared as static.
                             // Thus the containing class Car has the static aggregation
                             // relationship with theSpeed.
}
```

3.1.2.2 Automatic Aggregations

Aggregates that are declared without the *static* keyword are referred to as automatic aggregates. Hence, the containing class has automatic aggregation relationships with its automatic aggregates. They are often referred to as *instance variables*.

Note that the scope of the class `Windshield` must be non-static (declared with absence of *static* keyword), otherwise it is not a member class by definition.

```
class Car {
    public Engine theEngine; // Class Car has an automatic aggregation relationship with class Engine.
    private Tire theTires[]; // Class Car has an automatic aggregation relationship with class Tire.
    public Windshield theWindshield = "non-static" class Windshield { // class definition goes here};
                                // Class Car has an automatic aggregation relationship with the member
                                class Windshield
}
```

3.1.2.3 Implicit Inner-Class Aggregations (IIC)

For any inner class, there exists an implicit aggregate of the enclosing class (as described in Chapter 2). Hence, the inner class has an *implicit inner-class aggregation* relationship with its enclosing class.

```
class EnclosingClass {
    public class MemberClass {           // member class (non-static)
        private EnclosingClass this$0; //Class MemberClass has an IIC relationship with the
                                         //Enclosing class
        void fool { ....
            class LocalClass {           // local class
                private EnclosingClass this$0; //Class LocalClass has an IIC relationship with the
                                                //Enclosing class
            }
            MyAnonymousLocal foo2 { ...
                return new AnonymousLocal{ { // anonymous local class
                    private EnclosingClass this$0; }; //Class AnonymousLocal has an IIC
                                                        //relationship with the Enclosing class
                }
            }
        }
    }
}
```

It should be noted that the above example does not reflect the actual transformation of the source code that contains inner classes. Please refer to Chapter 2 for more details on inner classes. These variables are similar to instance variables in terms of scoping; however, they are more tightly coupled to the containing class since they are likely “used” by each other (i.e., bi-directional dependency).

3.1.3 Association Relationships

In [Kung et. al., 95a], an association relationship is defined as a complement of inheritance and aggregation relationships. In this model, we include the interface relationships as a part of the complement of the association relationship. We basically view the association relationships as relationships of a containing class with variables (associates) that are declared within a method body, including its signature. In an attempt to make the subject clear, we identify these associates as (1) local variables, (2) object parameter passing, (3) static variables, (4) local classes and anonymous classes (i.e., forms of inner classes).

3.1.3.1 Local Variables

Local variables are those declared well within a method's body (between { }). The class containing local variables may be data and/or message passing dependent on the local variables.

```
class C1 {
    ....
    void () {
        C2 c2 = new C2();;           // C1 depends on C2 via the local variable c2
        this.member_c11 = c2.member_c21;    // data dependence
        this.member_c11 = c2.getMember_c21(); // or via an accessor
        c2.foo();                          // message passing
    }
}
```

3.1.3.2 Object Parameter Passing

Object parameter passing refers to objects as formal parameters of method in a class. Formal parameters differ from local variables in that their declaration and value come

from outside the method's block (i.e., method's signature/declaration). We identify method parameters in the method declaration as follows:

```
class C1 {
    void f1(C2 c2, C3 c3,)           // c2 c3 are object parameters.
    {
        this.f(c2.do());               // C1 depends on C2 and C3
        if (this.member_c11 == c2.member_c21) // data dependence
            c3.foo();                   // message passing
    }
}
```

3.1.3.3 Static Variables

Static variables are variables whose members are declared with the *static* keyword. We can access these members without reference to any specific instance of these members' class.

```
class C1 {
    void f() {
        C2.foo();                       // C1 depends on static variable C2, C3
        if (this.isSth()) {              // message passing
            this.member = C3.Constant1; // data dependence
        }
    }
}
```

3.1.3.4 Local Classes – Anonymous Classes

Local classes are inner classes defined within a method. Anonymous classes are local classes declared without a class name. These variables are similar to local variables in terms of scoping; however, they are more tightly coupled to the containing class than local variables since they implicitly associate with the enclosing classes (i.e., likely bi-directional dependency).

```

public class C1 {
{
    void foo() {
        // The definition of a class as a local class.
        class C2 implements InterfaceI { // definition goes here }

        C2 c = this.new C2();           // C1 uses C2 within a method definition.
        this.member_c11 = c.a;         // data dependence
        this.member_c12 = c.foo();      // invoke static method of c type C2
    }
}

```

```

public class C1 {
{
    C2 foo(){
        // Here's the definition of a class as an anonymous class.
        return new C2{ // class definition goes here };
    }

    void test(){
        C2 c2 = foo();                 // C1 uses C2 here.
        this.member_c11 = c2.a;         // data dependence
        this.member_c12 = c2.foo();     // invoke a method of c2
    }
}

```

Note that this type of association relationships is only captured as long as the containing class actually uses the local classes/anonymous classes (i.e., their definitions don't count).

3.1.4 Interface Relationships

An interface relationship occurs whenever a class implements an interface. The interface relationship can be detected through Java's class declaration. The *implements* keyword specifies all interfaces implemented by the class being declared. Note that an interface cannot extend to a class.

```

class Car implements Jumpable - Class Car has an interface relationship with the Runnable interface.
class List extends Collection implements Iterator, Operator
    - Class List has interface relationships with the Iterator and Operator interfaces.

```

3.1.4.1 Strict-Data Interface Relationships

If an interface that a class implements does not contain any method signatures, then such an interface relationship is referred to as a *strict-data interface*. Otherwise, it is referred to as polymorphic (i.e., we have a polymorphic interface relationship). The following is an example of a strict-data interface.

```
Interface Jumpable {
    final int a = 1;           // optional "final" keyword indicates the attribute is constant
    String false = "FALSE";
}
```

Any class that implements Jumpable is considered related to the interface through strict-data interface relationship.

Having now defined the class relationships, the next step is to use them to construct a graphical representation of a program. To aid us in the latter, we make use of an ORG, which is discussed in the next section.

3.2 Object Relation Graph (ORG)

An Object Relation Graph (ORG) can be used as a graph representation of a class's firewall. An ORG contains a set of nodes, which are classes of the target system, and a set of edges, which categorize the inter-node relationships into four categories: inheritance (i.e., including polymorphic inheritance), aggregation, association, and interface. This paper will formalize the construction of the ORG in Java. The reference material in [Kung et. al., 95a] used to construct the ORG is modified to accommodate the new relationships that have been identified in Java.

Definition 1:

An edge-labeled digraph $G = (V, L, E)$ is a directed graph where $V = (V_1, V_2, \dots, V_n)$ is a finite set of nodes, $L = (L_1, L_2, \dots, L_n)$ is a finite set of labels, and $E \subseteq V \times V \times L$ is a set of labeled edges.

Definition 2:

The ORG for an OO program P is an edge-labeled digraph $ORG = (V, L, E)$ where V is the set of nodes representing the object classes in P , $L = \{Inh, Int, Agg, Ass\}$ is the set of edge labels, and $E \subseteq E_{Inh} \times E_{Int} \times E_{Agg} \times E_{Ass}$ is the set of edges defined below.

Definition 3:

$E_{Inh} \subseteq V \times V \times L$ is the set of directed edges representing the inheritance relationship between the classes. For any two classes $C_1, C_2 \in V$, $\langle C_1, C_2, Inh \rangle \in E_{Inh}$ indicates the C_1 *extends* from the class C_2 . The inheritance relationship includes the polymorphic inheritance refinement and interface inheritance relationship.

Definition 4:

$E_{Agg} \subseteq V \times V \times L$ is the set of directed edges representing the aggregation relationship between the classes. For any two classes $C_1, C_2 \in V$, $\langle C_1, C_2, Agg \rangle \in E_{Agg}$ indicates the C_1 contains one or more objects of class C_2 . The aggregation refines as automatic, static, and member class.

Definition 5:

$E_{Ass} \subseteq V \times V \times L$ is the set of directed edges representing the association relationship between two independent classes. For any two independent classes $C_1, C_2 \in V$, $\langle C_1, C_2, Ass \rangle \in E_{Ass}$ indicates the C_1 associates with C_2 in the following three ways:

- Class C_1 uses data members of class C_2 . This is called data dependency.
- Class C_2 's methods are invoked by some methods of class C_1 . This is called message passing.
- Class C_2 's objects are defined as formal parameters of member functions in class C_1 . This is called object parameter passing

where class C_2 is identified among local variables, static variables, local classes,

Definition 6:

$E_{Int} \subseteq V \times V \times L$ is the set of directed edges representing the interface relationship between the classes. For any two classes $C_1, C_2 \in V$, $\langle C_1, C_2, Int \rangle \in E_{Int}$ indicates the C_1 implements the interface C_2 . The interface relationship includes strict-data and strict-polymorphic refinements.

The ORG is used as a graphical representation of all classes and their relationships. The mechanism used to build the ORG is described in the following section.

3.3 Class Firewall in Java

A change can occur at different levels of a class, such as a data member change, a function member change, a class change, and a relation change. Each of these changes has a different impact on the modified class, and consequently on the system. A class's

firewall is a mechanism to identify the effects of a class change at the class level [Kung et. al., 94]. Changes to classes at the class level can be categorized in three types as follows (more details of class changes in other levels please refer to [Kung et. al., 94]):

Component changes	Change a defined/redefined method Add/delete a defined/redefined method Add/delete change local data	Affects states and transitions
“Interface” changes	Change an attribute access mode/scope Add/delete/change a defined / redefined datum Add/delete virtual abstract method	Affects operations and behavior
Relation changes	Add/delete super class Add/delete a subclass Add/delete an aggregated object Add/delete an association Add/delete an interface	Affects relationships between this class and others (including inner classes)

Figure 3-1 Class change identification at the class level and the effects

Note that the “relation changes” category has been modified to reflect the presence of the notion of the interface. We acknowledge that adding/deleting aggregated object and association (relation changes) may affect the relationships between the inner class and its enclosing class. The next section will complete the firewall theory with lemmas and proofs provided by [Kung et. al., 95a] along with some modifications to support our newly added relationship and refinements. We intend to provide complete lemmas and proofs (from Kung) to better comprehend the class firewall theory’s extension in Java. Readers should note that the extension is in plain text.

3.3.1 Class Firewall Theory

Intuitively, a class firewall for a class C , denoted as $CFW(C)$ in an OO program, is the set of classes that could be affected by changes to class C . These affected classes should be retested when class C is changed.

Lemma 1. *Let class A be a subclass of class B in the inheritance hierarchy, and only class B is changed without affecting its relationships with other classes. If the change affects the inherited members (from class B) of class A , for adequate testing, not only should class B be unit retested, but class A should also be retested and reintegrated with class B .*

Proof. *Since class A is a subclass of class B , class A must inherit some of class B 's attributes. Thus, class A has a code dependency on class B due to the inherited attributes. Because the modification made in class B affects the inherited members of class A , according to [Perry&Kaiser, 1990] they should be retested in class A to ensure that they work well in its reused context. Next, they should be reintegrated with other attributes of class A to make sure that they work well together. In addition, class A and class B should be reintegrated to make sure that correct member functions are executed in the inheritance hierarchy. This is necessary when polymorphism occurs since a request is sent to class A may be carried out by class B .*

Lemma 1.1. *Let class B be a superinterface of interface A in the (interface) inheritance hierarchy, and only interface B is changed without affecting its relationships with other classes/interfaces. If the change affects the inherited members (from interface B) of class A , for adequate testing, not only should interface B be unit retested, but interface A should also be retested and reintegrated with interface B .*

Proof. Since interface B is a superinterface of interface A, interface A may inherit some of interface B's attributes (i.e., constant values). Thus, interface A has data dependency on interface B due to the inherited attributes. Because the modification made in interface B affects the inherited members of interface, they should be retested in interface A to ensure that they work well in its reused context. Note that, in fact, interfaces are not normally subjected to explicit testing and due to polymorphism, the testing eventually involves classes that implement these interfaces.

Lemma 2. Let class A be an aggregate class of class B, and only class B is changed, without affecting its relationships with other classes. For adequate testing, not only should class B be unit retested, but class A should also be retested with class B.

Proof. Because a change of class B causes an effect on its object behavior or the operations of its member functions, class B should be retested at the unit level. Since class A contains some object of class B as its component, the change will affect the class A's object behavior in the following aspects:

- 1. The behavior of the aggregated part (e.g. class B's objects) is affected, and*
- 2. Other members of class A are affected if they are directly or indirectly dependent on the aggregated part.*

Thus, those dependent members of class A should be retested and reintegrated with the instances of class B. Moreover, class A's object behavior should be retested.

Lemma 3. Let A and B be two independent classes, and class A is associated to class B. If class A is changed without an effect on its relationships with other classes, then for

adequate testing, not only should class B be unit retested, but class A should also be retested and reintegrated with class B.

Proof. *Since class A is associated to class B, one or more of the following cases must occur.*

- *At least one member function of class A depends on some data members of class B,*
- *At least one member function of class A has a message passing to class B, and*
- *At least one member function of class A passes some class B objects as parameters in message passing.*

In all three cases, one or more members of class A have some data dependency on some of class B's members or objects. If the change in class B directly or indirectly affects those members or objects that the class A members are dependent on, then after class B retesting these dependent members of class A should be retested with class B. Later, these members should be reintegrated with other members of class A. Therefore, after changing class B, class A should be retested and reintegrated with class B to ensure adequate testing.

Lemma 4: *Let class A implement interface B, and only class B is changed without affecting its relationships with other classes. If the change affects the inherited attributes and/or methods' signature of class A, class A should be retested.*

Proof. *Analogous to the inheritance proof in Lemma 1, since class A implements interface B, the change in B will affect the class A in the following ways:*

- Class A may use some of class B's constant attributes, if any. Thus, class A has some data dependency on interface B.
- Class A must conform itself with respect to the changes in class B methods' signatures. These changes may involve updating/adding/deleting methods in class A.

Because the modification made in interface B affects the inherited attributes and methods of class A, class A should be retested.

To compute the class firewall, [Kung et al., 95] introduce a binary relation R that is derived from the directed edges of an $ORG = (V, L, E)$:

$$R = \{ \langle C_2, C_1 \rangle \mid C_1, C_2 \in V \wedge (\exists l \in L) \langle C_1, C_2, l \rangle \in E \}$$

Call R the dependence relation because it defines the dependency between the classes, according to the inheritance, aggregation, interface, and association relations. More specifically, $\langle C_2, C_1 \rangle \in R$ if and only if one of the following cases holds:

- C_1 is derived class of C_2
- C_1 implements interface C_2
- C_1 is an aggregate class of C_2 .
- C_1 is associated with C_2 by either accessing its data members or passing some messages.

In all of these cases, C_1 is dependent on C_2 in the sense that code changes to C_2 would affect the behavior of C_1 . The computed class firewall for a class C denoted $CCFW(C)$ then is defined as:

$$CCFW(C) = \{ C_j \mid \langle C, C_j \rangle \in R^* \},$$

where R^* is the transitive closure of R . That is, if $\langle C_i, C_j \rangle \in R$ and $\langle C_j, C_k \rangle \in R$, then and $\langle C_i, C_k \rangle \in R^*$. The transitive closure of R can be computed by the algorithm of [Aho et. al., 83].

The notion of the class firewalls can be extended to a set of changed classes. Let $S = \{C_1, C_2, \dots, C_k\}$ be a set classes that are changed. Then the class firewall for S , also denoted $CFW(S)$, is defined as follows:

$$CFW(S) = \bigcup_{\forall C \in S} CFW(C)$$

The complete theorems/proofs of the class firewall construction can be found in [Kung et. al., 95a].

3.3.2 Discussions on the Implementation of the Class Firewall Algorithm

The transitive closure algorithm, Warshall's in [Aho et. al., 83] recommended by Kung, is an intuitive way of describing impact analysis and is straightforward in terms of implementation [Bohner&Arnold, 96]. However, this algorithm with its running time of $O(n^3)$, where n is the number of nodes (classes), is considered slow. Some faster algorithms have been developed since then - a transitive closure based on a boolean matrix presentation can save time and space since the algorithm uses logical operations on bit values (boolean) rather than integer values. Furthermore, in practice, it is desirable to obtain only the set of affected classes when changes are introduced in one or more classes, and not to specifically be concerned with which classes affect what. To that end, transitive closure is more than sufficient, especially where the number of modified classes is small. Bearing these two aspects in mind, we believe that there is no need to implement a full-fledged transitive closure algorithm, in particular computing

the entire dependencies up front. Instead, we suggest a more efficient approach. Specifically, our implementation through the Depth-First Search algorithm (DFS) is sufficient to compute the class firewall of a modified class (or classes) without resorting to transitive closure. Indeed, given a node (or nodes), the DFS computes the set of nodes that are reachable from the given node (or nodes). With respect to a class firewall, DFS correctly derives all classes affected by changes to one or more classes. The running time of this DFS algorithm is $O(n + e)$ or $O(n^2)$. Another advantage of this algorithm is that it is adaptable to compute the class firewall upon request rather than up-front. The following is a modified version of DFS found in [Cormen et. al., 90] to compute a class firewall for a set of selected nodes.

```

Input: ORG=(V,L,E)
Output: boolean indicate cycle
 $\pi[u]$  = predecessor or parent of u.
 $f[u]$  = finishing time of u.
color[u] set the color for u.
boolean ClassFirewall(ORG, selectedNodes) – A modified version of [Cormen et. al., 90]’s
For each vertex  $u \in \text{selectedNodes}$ 
1) color[u] = WHITE;
2)  $\pi[u]$  = null, // store predecessor i.e., parent
3) time = 0; // store finishing time (global)
4) cycle = false; // store cyclic boolean (global)
5) for each vertex  $u \in V$ 
6) do if color[u] == WHITE
7) DFS_Visit(u);

DFS_Visit(u)
1) color[u] = GRAY; // visiting
2) d[u] = time = time + 1;
3) for each  $v \in \text{Adj}[u]$ 
4) do if color[v] = WHITE
5)  $\pi[v]$  = u;
6) DFS_VISIT(v);
7) else if color[v] = GRAY // visit its ancestor
8) cycle = true; // we found a cycle for later use
9) color[u] = BLACK; // visited
10)  $f[u]$  = time = time + 1; // advance the finishing time

```

Figure 3-2 The class firewall algorithm - a modified version of DFS [Cormen et. al., 90]

Let us explain the algorithm. Basically, the DFS algorithm works by selecting one CUT node at a time (i.e., in the *selectedNodes*) as a start node; the CUT is marked as *visiting* (i.e., set its color to GRAY). Then each *unvisiting* (i.e., color WHITE) neighbour whose edge is incident⁷ to the CUT will be explored in a recursive fashion (line 6 in the *DFS_Visit()*). Once the *visiting* node cannot reach any other neighbour (all *visited* or no neighbour), the *visiting* node is then set to *visited* (i.e., set the color to BLACK and advance the finishing time *f[u]*) and the process backtracks to the node, which has been visiting immediately before this node. When all nodes that are reachable from the CUT, are *visited*, the class firewall for that CUT is complete. If any CUT node in the *selectedNodes* remains unvisited, the process will select that CUT node as a new start node for another *DFS_Visit()* quest. This process is repeated until all CUTs have been visited. The set of nodes whose colors are BLACK is indeed the class firewall for a given set of CUTs.

We revisit the example of the class firewall in Chapter 1, applying the above version of the DFS algorithm (figure 3-2) on the Builder class. The main *ClassFirewall* method accepts the entire ORG and the Builder as its input. After the initialization (line 1-5 in the main), the *DFS_Visit()* will be applied on the Builder. The Builder node is marked as *visiting* (line 1 of *DFS_Visit()*), and then the process tries to apply the same routine on each of the *unvisiting* adjacent nodes of the Builder (line 6 of *DFS_Visit()*). In this particular case, it only finds the adjacent Vehicle. By the same token, the Vehicle node is marked as *visiting* and the search continues on the Vehicle's neighbors, which could

⁷ We follow the direction of "affects" instead of "depends on".

be, for example, lined up in order of Motorcycle, Car, and Driver nodes. After each of the neighbors of Vehicle are searched and visited (i.e., set the color to BLACK and advance the finishing time $f[u]$) in a recursive fashion, the process backtracks to the Vehicle. Since the Vehicle has no more *unvisited* neighbours (whose colors are WHITE), it is marked as *visited*, the process resumes to the Builder. Similarly, the Builder is marked *visited* and the process backtracks to the main *ClassFirewall*. Since there is no other *unvisited* node in the *selectedNodes* set, the process terminates with 5 nodes visited in order of the finishing times: Driver, Car, Motorcycle, Vehicle, and Builder. And this set is indeed the class firewall for the CUT Builder.

Now assume the Tire node is added to the CUT set. The process continues by returning to where it left for computing the class firewall of the CUT Builder node. Now, the DFS selects the Tire node as new node and applies the *DFS_Visit()* on it. When the process looks for the Tire's neighbors, it finds out that the Vehicle is already visited (by the Builder). The process then marks the Tire node as *visited*, and resumes to the main program. Finally, the class firewall of the CUT {Builder, Tire} is computed as {Motorcycle, Car, Driver, Vehicle, Builder, and Tire}. Figure (3-3) depicts the class firewall for the CUT {Builder, Tire}.

Figure 3-3 The class firewall of the CUT Builder and Tire

It is worth noting that the algorithm is inherently incremental in that when a node, say Tire, reaches a previously visited node, say Vehicle, the Tire does not bother to compute its dependencies along the Vehicle's edges, but the Tire simply "includes" all dependencies of Vehicle including Vehicle itself in the set of its dependencies. And hence, it guarantees that this algorithm visits every node and every edge at most once. Thus, the class firewall algorithm based on DFS can acquire in $O(n + e)$ time, where n is the number of nodes and e is the number of edges of the ORG input.

3.4 Testing Order

Once the class firewall for a modified class is computed, testers should perform class unit testing followed by integration testing [McGregor&Korson, 94] for all classes enclosed in the resulting firewall [Lueng&White, 92]. The authors point out "[these steps] should work in concert on finding faults which cannot be easily detected by other

phases. At the same time, for faults which can be detected by several phases, resources should not be expended in duplicating the same test effort". The term "resources" refers to constructing test stubs and test drivers in general. It is costly and difficult to construct such test stubs and drivers for OO programs for a number of reasons [Kung et. al., 95a]. They explain " [f]irst, simulating a class or its objects is costly and difficult because the tester not only has to understand and simulate its class members, object states, and behaviors, but also has to understand and simulate its inherited members and component class objects. Second, simulating a member function is expensive and complicated because the tester has to understand chains of member function invocations and simulate the different state changes of class objects and messages between the class object due to these function invocations to construct appropriate test stubs". The authors suggest that if we can test components using tested components by following a proper test sequence, then test efforts for constructing test stubs and test drivers can be reduced. The main idea is to test the independent components first and then test the dependent components based on their relationships. For example, a superclass should be tested before its subclasses so that the stub and the information for testing the superclass can be reduced and reused for testing its subclasses respectively. They introduce a test strategy, called *test order*, which generates a cost-effective test sequence for class unit retesting and class reintegration testing for C++ programs. The test order algorithm proposed by [Kung et. al., 95a] is presented next.

3.4.1 Overview of Kung's Test Order Algorithm [Kung et. al., 95a]

A test order of the given class firewall is basically a result of a topological sort using the dependency relation as the precedence relation on the class firewall. However, the topological sort cannot perform on a cyclic graph, which is likely to occur. To overcome this hindrance, the graph must transform to be acyclic so that topological sorting can be performed. Kung suggests that the cyclic $ORG(V, L, E)$ (or class firewall) can be converted into an acyclic digraph $ORG'(V', L', E')$ using the notion of clusters or strongly connected components (SCC). A **cluster** is defined as is a maximal set of nodes that are mutually reachable through the relation R . A cyclic ORG may have more than one cluster, and a cluster may contain a minimum of one vertex. Clusters that contain only one vertex are referred to as unit clusters, otherwise non-unit clusters. Treating each cluster as a node results the digraph ORG' in acyclic. Otherwise, some of the clusters must not be a maximal set of mutually reachable vertices. Applying topological sorting on the entire ORG' results in the *major* test order. For each cycle of the non-unit cluster, edges will be removed one at a time until the cycle breaks. Applying topological sorting on resulting acyclic clusters produces the *minor* test orders. The cyclic ORG (or class firewall) can be tested first according to the major test order and then the minor order.

The outline of the test order algorithm:

Step 1:	Transform the ORG into an acyclic digraph ORG .
Step 2:	Produce a topological sorting for the ORG' . The test order produced is called the major test order.
Step 3:	For each non-unit cluster of ORG' do steps 4 to 5.
Step 4:	For each cycle of the cluster, select and remove an edge to break the cycle.
Step 5:	Produce a topological sorting for the acyclic subdigraph obtained in step 4. The test order produced is called the minor test order.

Figure 3-4 The outline of the test order algorithm [Kung et.al., 95a]

The next two sections discuss the algorithm to convert a cyclic ORG to an acyclic one, and rules for breaking cycles in a cluster.

3.4.2 Converting a Cyclic ORG to an Acyclic ORG Using Clusters [Kung et. al., 95a]

Clusters are computed by using the transitive closure of the dependency relation R defined in the Chapter 3. For any two vertices $u, v \in V$, where V is the vertex set of the original cyclic $ORG(V, L, E)$, $\langle u, v \rangle \in R$ means a code change in u would affect the behavior of v , and $\langle u, v \rangle \in R^*$ means v is transitively affected by a code change in u . Clearly, if $\langle u, v \rangle \in R$ and $\langle u, v \rangle \in R^*$, then changes to u affect v and vice versa. In this case, u and v are mutually reachable and they are in a cycle. Therefore, they must be in the same cluster. The purpose of this algorithm is to identify such pairs and place them in a cluster and compute the edges that define the precedence relation between the clusters (resulting in a major test order). The complete outline of the algorithm can be found in [Kung et. al., 95a].

3.4.3 Cycle Breaking

It was mentioned earlier that cycles in a cluster can be broken by temporarily removing the edges of the cluster one at a time. In [Kung et al., 95], it is asserted that any cycle must contain at least an association edge and association edges represent the weakest coupling between two related classes. The authors suggest an implementation that

simply picks up any association edge and removes it until the cycle breaks. However, there is no reference on how it is accomplished.

3.4.4 Analysis on the Complexity of Kung's Test Order Algorithm.

We now briefly consider the complexity of the entire test order algorithm (based on the outline in Figure 3-4). First, the “acyclic-converting” algorithm is examined (in step 1). Since the transformation is entirely based on the transitive closure algorithm to compute clusters and since the recommended transitive closure computation is accomplished in $O(n^3)$ time (as discussed in section 3.3), the complexity of the “acyclic converting” algorithm is $O(n^3)$. Second, the computation for the topological orders (in step 2 and step 5) are presumably bound to $O(n^2)$ time. And because the algorithm for cycle breaking is not presented, we may deduce that the complexity of the entire test order is at least⁸ bound to $O(n^3)$ time. The following section describes an alternative algorithm for test order algorithm.

3.5 An Alternative of the Test Order Algorithm

Our proposed test order algorithm basically follows the outline of the algorithm from Kung (see figure 3-4). The key difference between our proposed algorithm and the one from Kung lies in the computation of clusters (step 1 of figure 3-4). Moreover, our solution makes extensive use of the DFS algorithm as its primary tool to traverse the ORG.

The DFS is able to compute the topological sort [Cormen et. al., 90]. The reason we choose the DFS is because it is used across the entire class firewall model, and more importantly, it produces useful information about the graph structure. For instance, some of the valuable information produced include cycle indicators and the graph structure resulting from its traversal (i.e., depth-first trees). The former indicates whether clusters exist and the latter indicates whether two or more nodes are in the same cluster (i.e., two nodes are in the same cluster if they are in the same depth-first tree after the completion of SCC algorithm, which is described later on).

Unlike the original test order version, which used the transitive closure to compute clusters, we instead use the strongly connected component algorithm (SCC) found in [Cormen et. al., 90]. According to the SCC algorithm, clusters can be computed by applying two DFSs, once on an ORG and once on the transpose of this ORG, denoted ORG^T . The acyclic ORG^T can be achieved by shrinking each cluster into a single node. The major test order of the acyclic ORG and the minor test orders of each cluster in ORG^T or ORG derive the desired test order. For more details on SCC algorithm and related information, readers are invited to consult [Cormen et.al., 90].

The simple implementation for breaking cycles again is based on the DFS to break cycles in clusters by temporarily removing a set of edges.

⁸ "at least" implies that the unknown complexity of the cycle breaking algorithm may be larger than $O(n^3)$

The following is the outline of the alternative test order algorithm.

The outline of the alternative test order algorithm

Step 1. Produce a topological sorting for ORG. If the ORG is non-cyclic then terminates.

The list of finishing time $f[u]$ of each node u in decreasing order is the test order. Otherwise,

Step 2. Transform ORG to acyclic by using SCC algorithm [defined in the figure (3-7)].

Step 2.1 Transform ORG to ORG^T .

Step 2.2 Produce a second DFS on ORG^T . It correctly computes clusters.

Step 3. Produce a topological sorting on "acyclic" ORG. The result is the major test order.

Step 4. Breaking cycles in each non-unit cluster.

Step 5. Produce a topological sorting on the resulting clusters generated in step 4.

The result is the minor test order.

Figure 3-5 The outline of the alternative test order algorithm

3.5.1 Overview of the Alternative Test Order Algorithm

The new algorithm is basically composed of five steps. Given an ORG, the algorithm in step 1 will perform a topological sort, a variation of DFS found in [Cormen et. al., 90], to attempt to compute the major test-order. At the same time, it also determines whether the ORG is cyclic or not. If it is not, the algorithm will terminate and the major test order can be listed in the decreasing order fashion of the finishing time of nodes. The finishing time of a node indicates that all its neighbors have been visited completely (similar to mark the node as *visited*). In case of existing cycles, the strongly connect components algorithm (SCC), which replaces the transitive closure algorithm, will assist us in computing the clusters. Recall that clusters can be computed by applying two DFSs, once on an ORG and once on the transpose of this ORG. The acyclic ORG^T can be achieved by collapsing each cluster into a single node. The process is listed in step 2 of the outline algorithm. Analogous to the original version, once all clusters are

identified, topological sorts can be performed to find the major test order and minor test order as in step 3 and 5 of the main algorithm. The final test order can be analogously arranged like that of the original version. Figures (3-6) and (3-7) summarize the original version of SCC and the topological sort algorithm [Cormen et. al., 90].

Topological-Sort(G)

- Step 1: Call DFS(G) to compute finishing times $f[v]$ for each vertex v .
- Step 2: As each vertex is finished, insert it onto the front of the linked list.
- Step 3: Return the linked list.

Figure 3-6 The topological sort algorithm [Cormen et. al., 90]

Strongly-Connected-Components (G) [Cormen et. al., 90]

- Step 1: Call DFS(G) to compute finishing time $f[u]$ for each vertex u .
- Step 2: Compute G^T .
- Step 3: Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing of finishing time. Vertices of each tree in depth first forests are separate strongly connected components.

Figure 3-7 The Strongly Connected Components algorithm [Cormen et.al., 90]

3.5.1.1 An Example of How Topological Sort Produces a Test Order

We now take a quick look on how topological sorting can generate a test order for the previous example [see section 3.3]. Recall that the class firewall of the Builder and Tire is {Driver, Car, Motorcycle, Vehicle, Builder, Tire}, which is in the order of the finishing times or *visited*. We know that a test order is in the decreasing order of the finishing times. Thus, a test order could be {Tire, Builder, Vehicle, Motorcycle, Car, Driver}. As can be visually detected, the computed class firewall has no cycles so one DFS is necessary to compute a test order. The Builder and Tire classes are independent,

so they should be tested first. The Vehicle class can be immediately tested after the Builder and Tire classes are tested since it only depends on the two classes. Then, the class Driver, Car, and Motorcycle can be sequentially tested. Thus, the topological sort algorithm correctly computes a test order for the given class firewall. The following section will focus on breaking cycles strategy in Java.

3.5.2 Cycle Breaking in Java

This section focuses on how cycles of a cluster can be broken in Java. We first prove that the interface relationship does not constitute a cycle. Secondly, we describe another form of cycle existing in Java and propose a new cycle breaking rule, an extension of the original one. Finally, we propose our algorithm.

3.5.2.1 Interface Relationship Does Not Involve in Cycles

Since Java introduces the interface relationship, it raises the issue as to the impact of this relationship on the approach to cycle breaking presented in the previous section. To answer this question, we need to determine the coupling of classes under the interface relationship. We found that the interface relationship represents the weakest form of relationship between two classes (i.e., more precisely, a class and an interface) by the following observation: if the interface contains data, the interface relationship involves only data coupling. Otherwise, they are independent.

Thus, in general, the most that can be observed is that the interface relationship possesses data coupling. And since data coupling is weaker than control coupling

[Page-Jones, 80], the interface relationship presents the weakest form of coupling. More importantly, we also observe that the addition of interface relationships does not produce cycles. The proof is based on a single observation: there are no edges coming out from an interface except the ones to its superinterfaces, and these (interface inheritance) edges are only uni-directional. The proof proceeds as follows:

Assume that a cycle contains an interface edge $\langle v_n, v_0 \rangle$ in which v_n is a class node and v_0 is an interface node. Since it is cyclic, there exists a path connecting v_0 back to v_n , say $(v_0, v_1, v_2, \dots, v_{n-1}, v_n, v_0)$. Because v_0 is an interface node, the edge that connects v_0 to v_1 must be an (interface) inheritance edge (i.e., v_1 class is a superinterface of v_0 class) by definition of interface in Java (an interface can neither extend to a class nor use any other Java objects). Thus, v_1 is an interface node. The same argument can be applied to $v_2, \dots, v_n$. This contradicts the fact that v_n is a class. Thus, there can be no interface edges appearing in an ORG cycle.

3.5.2.2 Cycle Breaking Rule in Java

The fact that every cycle has one or more association edges no longer holds in Java. In fact, we discover another form of cycle that is made up by only aggregation edges. More precisely, this type of cycle is uniquely formed of only two aggregation edges, one of type *implicit inner-class aggregation* and the other of type *automatic aggregation* of the enclosing class. We refer to this type of cycle as the *inner-class aggregation cycle* (IAC). The following example illustrates this type of cycle.

```

class EnclosingClass {
    int a = 10;
    private InnerClass theInner = new InnerClass ();
    // The Enclosing has an automatic aggregation relationship with its Inner Class.
    void advance(){
        theInner.advance();
    }
    class InnerClass {
        EnclosingClass this$0; // InnerClass has an implicit aggregate of the enclosing class. The
        // Inner Class has an implicitly enclosing class aggregation with
        // its EnclosingClass.
        void advance() { this$0.a++ };
    }
}

```

Figure 3-8 A sample of code demonstrating a newly found type of cycles, the inner class aggregation cycle

The ORG of the above example contains an inner-class aggregation cycle as the

Figure 3-9 The formation of the inner-class aggregation cycle

following figure (3-9):

The proposed breaking cycle rule in Java is as follows:

- If the detected cycle is of type inner-class aggregation, removing either edge will break the detected cycle (i.e., removing the *(automatic) aggregation* edge).
- Otherwise, applying the old cycle breaking rule. Removing any association edge until cycles are broken.

3.5.3 The Cycle Breaking Algorithm

Our cycle breaking strategy is to simply remove (1) all (automatic) aggregation edges involving in the *inner-class aggregation cycle* and (2) all association edges that may be involved in any cycle of a cluster by using one DFS pass. The former guarantees that removing one edge will break the two-edged *inner-class aggregation cycle* into an acyclic component. Regarding the latter, as we know, during the DFS traversal of a digraph, there are four different types of edges that can be found. The *tree*-edges are edges leading to new nodes in a depth-first forest. The *back*-edges are ones indicating the existence of a cycle since they are connected to their *visiting* ancestors (whose color are GRAY). The other two are *forward*-edges and *cross*-edges, which are non-tree edges. To put it simply, any cycle detected by DFS basically consists of tree-edges and a back-edge. Our algorithm removes aggregation edges constituting tree-edges in the depth-first forest. And hence, all cycles are broken. It is done by forcing the process to visit a new node through non-association edges (tree-edges). If association edges are the only edges connected to a new node, these edges are logically removed. The modified version of DFS to break cycles can be depicted as the following figure (3-10).

```

boolean CycleBreaking(cluster) – A modified version of DFS to break cycles
1) For each node, the set of association edges must be ordered at the end
2) For each vertex  $u \in \text{cluster}$ 
  (3) do if color[u] == WHITE
    (4) DFS_Visit(u);

DFS_Visit(u)
  1) color[u] = GRAY;
  2) for each  $v \in \text{Adj}[u]$ 
    3) do if color[v] = WHITE
      4) if (edge(u,v) is an association edge || edge(u,v) is an automatic agg. of IAC)
        5) remove edge(u,v) // simply marks it removed
      6) else DFS_VISIT(v);
    7) If color[v] = GRAY
      8) remove edge(u,v) // detecting a cycle - simply marks it removed
  9) color[u] = BLACK;

```

Figure 3-10 The outline of a modified DFS algorithm to break cycles in a cluster

3.5.4 Analysis of the Correctness of the Alternative Test Order Algorithm

The correctness of the proposed algorithm is a consequence of the correctness of the original version, of the strongly connected component, and of the cycle breaking algorithm. The proofs for the first two algorithms can be found in [Kung et. al. 95] and [Cormen et. al., 90], respectively. What is of interest is to ensure that the topological sort does indeed produce a precedence relation order and the correctness of the breaking cycle algorithm. The key to proving the correctness of topological sort to produce the test order relies on how to determine the finishing time of any two nodes u, v along the edge (u, v) . We need to prove that for any edge (u, v) the finishing time of u must be *smaller* of that of v (i.e., $f[v] < f[u]$). In terms of a class firewall, if class u depends on class v , then u should be tested *after* v . Let us consider an edge (u, v) that is *visiting* by DFS and where v cannot be gray. If v is white, v becomes a descendant of u (i.e., u is *visiting* before v), then $f[v] < f[u]$. If v is black, then $f[v] < f[u]$ because u was already *visited* before v . Thus, for any edge (u, v) in a directed graph, $f[v] < f[u]$. Returning to the proposed test order algorithm, if the ORG is acyclic, then the topological sort correctly produces the test order in step 1, otherwise topological sorts in step 3 and 5 correctly produce the major and minor test orders.

To prove the correctness of the proposed breaking cycle algorithm, we only need to prove that no tree-edges in the depth-first forest produced by the algorithm are association edges. Recall that there are 4 types of edges found in an ORG. The line (1) in the main DFS algorithm ensures the DFS will not use any edges of type association

to connect a node to a new node unless there is no choice. That would reduce the number of association edges to be removed by taking advantages of the nature of DFS traversal, the preorder traversal. The traversal likely pushes the association edges to be *forward*-edges or *cross*-edges or back edges. In case of association edges to be tree-edges, these edges will be filtered out at line (3) in the DFS_Visit() method. And this check enforces the correctness of the algorithm.

3.5.5 Analysis of the Complexity of the Alternative Algorithm

To put it simply, the DFSs and the transpose algorithm constitute the execution time of the proposed algorithm for test order algorithm. First, the execution time of DFS is bound to $O(n + e)$ (mentioned in section 3.3.2). Second, the transpose on G will visit every vertex and its adjacent edges once for reversing the direction; thus, the running time is also $O(n + e)$ under the assumption we use adjacency-list representation of the graph. Therefore the total running time of the testing order algorithm is $O(n + e)$ or $O(n^2)$. We can conclude that the proposed implementation of the class firewall and the proposed test order algorithm are simpler and more efficient than those using direct transitive closure algorithm as in [Kung et. al., 95a]. However, we also recognize that our solution can be improved in terms of reducing the number of removed association edges.

3.6 Discussion on the Effectiveness of the Class Firewall Strategy

Now the only remaining question is how to determine the effectiveness (cost benefits) of the class firewall strategy. To put it differently, when testers have already computed test firewall and/or test order, they want to know whether they should use the information produced by class firewall strategy to conduct their regression testing or they just perform the “retest-all” strategy. The key answer probably relies on the size of the class firewall and the effectiveness of the test order. It is obvious that if the firewall closure is relatively small, the firewall strategy is definitely favourable. Unfortunately, this is not always the case, especially when polymorphic dependencies are accounted [to be discussed in Chapter 5]. So the strategy may not be useful in such case. It is more interesting to note that, in [Kung et. al., 95a], the authors conduct a test to see how useful the test order is by unit testing their InterView library 200 times with random orders and one with computed test order. The result is quite convincing: an average of stub needed is 192.3 for each random testing versus 8 for testing using test order as the road map. Thus, we may deduce that the class firewall strategy is always more effective than the “retest-all” strategy. For a concrete approach to determine to use selective regression testing strategy (i.e., the class firewall), readers are invited to consult [Lueng&White, 95]⁹.

⁹ They present a test cost model and identify the conditions under which the selective strategy is more economical than the “retest-all”.

Chapter 4 - Detailed Examples

This section provides two examples of obtaining a class firewall and the test order in detail. The first example, a banking example, introduces a complete complexity of the impact analysis in Java. The banking system is sophisticated enough to cover all aspects of the class firewall model, including the cycles issue. The second example demonstrates the actual strength of JFireClass by using the tool to perform IA on a commercial Java product, JGL from ObjectSpace Inc. However, the use or correctness of JGL is not of interest to this paper.

The format of the examples is as follows. First, the *UML Class Diagram* will be manually modeled from the given source code. Second, the ORG presentation generated by JFireClass is provided. JFireClass effectively identifies four main categories of class relationships: inheritance, interface, aggregation, and association. Third, the class firewall is computed for a given CUT. Finally, the test order is generated for the affected classes and/or for the entire system.

Before examining these examples, we will present the UML notation supported in JFireClass. Because JFireClass analyses the effects of change(s) introduced in a class based on class relationships, the *Class Diagram* is an appropriate tool to report the findings of the tool. The following section shows the reader the set of UML notation is used throughout the examples.

4.1 UML Notation in JFireClass

4.1.1 Class Structure

A class structure is rendered as a rectangle with three parts: name, attributes, and methods, as follows:

Figure 4-1 A full presentation of a class

Figure 4-2 A short presentation of a class

The short form of a class structure only contains the class header. By default, JFireClass generates the short forms. The full format is displayed upon request.

JFireClass represents an interface slightly differently than UML since it must reveal the attribute parts. Both the full form and the short form of an interface are shown:

Figure 4-3 A full presentation of an interface

Figure 4-4 A short presentation of an interface

Our class structure supports (1) levels of visibility: public (+), protected (#), private (-), (2) scope: instance, classifier (i.e. static variables will be underlined). Local variables are optionally included in the method part.

4.1.2 Class Relationships

For class relationships, we support the following notation:

UML (standard)	UML in JFireClass
Generalization	Inheritance
Association Aggregation Composition	Aggregation
Dependency	Association
Realization	Interface

Although the first three relations are named differently, they are equivalent in their meaning respectively. For aggregation, we omit the multiplicity and the difference in semantics between the aggregation and composition relation because they are pure semantic intents and are likely impossible to capture [see Chapter 2]. It should also be noted that the realization relationship in standard UML includes not only the notion of an interface but also other concepts such as use cases and collaborations.

4.2 Bank Example

The Bank example is aimed at offering the reader a deeper understanding of the class firewall model. The example comes with ten classes including one interface. The bank example allows customers to perform some basic transactions such as creating a new account, and requesting a deposit or a withdrawal. We attempt to illustrate all aspects of the theory with this small package so that we can understand not only the theory but also the capability of JFireClass.

4.2.1 The Class Diagram and the Corresponding ORG of the Bank Example

The complete class diagram of the Bank example follows:

Figure 4-5 A Class diagram for the Bank Example

The equivalent ORG is constructed in JFireClass as the following figure 4-6:

Figure 4-6 An ORG representation for the Bank Example constructed from JFireClass

4.2.2 The Class Firewall of the CUT BankAccount

We firstly take a look at the class firewall of the BankAccount class generated by JFireClass in the following figure (4-7):

Figure 4-7 The class firewall of the BankAccount class

JFireClass denotes CUT nodes by shading them in the darkest color; affected class nodes, in lighter dark color. As shown in Figure (4-7), Portfolio, Portfolio\$PortfolioEnumerator, Bank\$BankEnumerator, Bank, Preferred-CustomerCheckingAccount, CheckingAccount, and SavingAccount are reachable from, equally affected by, the BankAccount class node. Thus, they together with the BankAccount enclose the class firewall of the BankAccount class.

Now we look into the anatomy of the class firewall process. Similar to the Car example in Chapter 3, the class firewall algorithm starts on the CUT node; BankAccount is marked as *visiting*. Then, DFS_Visit() is recursively called upon the currently visiting node's neighbors, (Portfolio, Bank, CheckingAccount, SavingAccount), whose edges are incident to it. The process visits Portfolio then Portfolio\$PortfolioEnumerator, the only Portfolio's neighbor. Since Portfolio\$PortfolioEnumerator has no unvisited neighbor (whose color is BLACK), the process marks the node as *visited*, advances the finishing times and backtracks to Portfolio. Having found out that Bank is the remaining Portfolio's neighbor, the process continues its search on the Bank. By the same token, the process recursively visits Bank, and its reachable nodes. The Bank\$BankEnumerator and Bank are then visited in order of the finishing times respectively. The process then completes the search on Portfolio; it marks Portfolio as *visited* and backtracks to BankAccount. The process then skips the search on the Bank class node since it is already visited by the search on Portfolio's path. CheckingAccount and SavingAccount, the remaining neighbors of the CUT, are then searched in the same fashion as Portfolio is. Finally, the class firewall initiating from the BankAccount

encloses {Portfolio\$PortfolioEnumerator, Bank\$BankEnumerator, Bank, Portfolio, PreferredCustomerCheckingAccount, CheckingAccount, SavingAccount, BankAccount} in order of the finishing times (or *visited*).

4.2.3 Test Order of the Class Firewall of the CUT BankAccount

This section leads us to the test order of the firewall originating from the BankAccount by exposing the various intermediate results of the alternative test order algorithm. First of all, the DFS (as in the first line of the outline of the alternative test order algorithm, fig. 3-5) will run on the entire graph to detect if any cycle exists. We again start the process from the CUT, BankAccount. The result confirms that the model is indeed cyclic and the affected nodes will be rearranged in the decreasing order of the finishing times. To be clear, the cycle is detected when the process visiting the Portfolio\$PortfolioEnumerator node. The node finds out that its neighbor, the Portfolio, is also its ancestor (i.e., color GRAY), and is revisited by its descendant (as in line 7 of the DFS_VISIT() in Figure 3-2). The cycle flag is then set. Finally, the algorithm produces the arrangement of the affected class nodes in the decreasing order of the finishing times: {BankAccount, SavingAccount, CheckingAccount, PreferredCustomerCheckingAccount, Portfolio, Bank, Bank\$BankEnumerator, Portfolio\$PortfolioEnumerator}. The following figure (4-8) depicts the current process.

Figure 4-8 The application of the first DFS on the Bank's ORG

Because the ORG graph is cyclic, it must undergo the transpose operation. The second DFS() will be applied on the new transpose graph with the order of the resulting arrangement from the first DFS (as in line 2 of the outline algorithm in figure 3-3). This DFS() produces strongly connected components, each of which is a depth first tree by the completion of running DFS_Visit() on a particular node. At the same time, it also produces the major test order. The following figure 4-9 illustrates the outcome of the second DFS to produce clusters equivalently and the major test order.

Figure 4-9 Production of clusters from the second DFS on the transpose ORG

In this example, the algorithm determines that there are two non-unit clusters: {Portfolio, Portfolio\$PortfolioEnumerator} and {Bank\$BankEnumerator, Bank}. As shown in the above figure, the major orders remain the same values within a cluster.

Let us apply the cycle breaking algorithm on these two clusters. The *CycleBreaking* algorithm works similarly to the original DFS algorithm. Let us process the {Portfolio, Portfolio\$PortfolioEnumerator} cluster first. The algorithm accepts this cluster as input and selects the Portfolio as the start node to begin the search. The process marks the Portfolio node as *visiting* and attempts to visit each unvisited neighbor along edges whose types are neither of association nor aggregation involved in the *implicitly inner class* cycle. In this case, the process finds the edge is a “valid” *implicitly inner aggregation* and visits the Portfolio\$PortfolioEnumerator. Since Portfolio\$PortfolioEnumerator has no unvisited neighbor, the process marks the

Portfolio\$PortfolioEnumerator as *visited* and returns to the Portfolio and terminates. Finally, the {Portfolio, Portfolio\$PortfolioEnumerator} cluster becomes an acyclic digraph by removing one aggregation edge (i.e., an automatic aggregation edge of an *implicitly inner class* cycle). And the minor test order of the cluster can be arranged as {Portfolio, Portfolio\$PortfolioEnumerator} with respect to the decreasing order of the finishing times.

Similar to the {Portfolio, Portfolio\$PortfolioEnumerator} cluster, the *CycleBreaking* on {Bank, Bank\$BankEnumerator} starts off with the Bank node; it marks the Bank as *visiting* and attempts to apply the same process to its neighbor. This time, the process finds out that the incident edge from Bank\$BankEnumerator is a “valid” aggregation edge; therefore, the process visits the Bank\$BankEnumerator. The process marks Bank\$BankEnumerator as *visited* and backtracks to the Bank and finally terminates. The minor test order of the {Bank, Bank\$BankEnumerator} cluster can be arranged as {Bank, Bank\$BankEnumerator} with respect to the decreasing order of the finishing times. The final test order is: {BankAccount, SavingAccount, CheckingAccount, PreferredCustomerCheckingAccount, Portfolio, Portfolio\$PortfolioEnumeration, Bank, Bank\$BankEnumerator} as shown in figure (4-10):

Figure 4-10 Production of a minor test order from the last DFS

The following figure (4-11) shows the same test order result for the BankAccount firewall as was generated from JFireClass.

Figure 4-11 A test order of the BankAccount firewall generated by JFireClass

Figure (4-12) below depicts the test order of the entire Bank Example. Testers may run this test prior to the first time of constructing the test stub and drivers to save their test effort.

Figure 4-12 A test order of the entire Bank Example generated by JFireClass

4.3 JGL Example

ObjectSpace's JGL offers a library of data structures classes (i.e., lists, stacks, etc.) to store other objects, as well as a set of algorithms (i.e., sorting) to manipulate those objects. ObjectSpace made JGL free for commercial use. JGL 3.1 contains 173 classes in seven packages (excluding the Voyager library). JGL 3.1 is our target for demonstrating the strength of JFireClass. This section will show the ORG presentation of JGL, the class firewall on a certain class, and finally the test order of the class

4.3.2 The Class Firewall of JGL Sample for the CUT ForwardIterator and Test Order of the Class Firewall

The ForwardIterator's class firewall and test order follows:

Figure 4-14 The class firewall and test order of the CUT ForwardIterator generated by JFireClass

These figures (4-13 & 4-14) demonstrate that our tool does scale up to handling a real example.

4.3.3 Some Statistical Figures on the Execution Time of JFireClass on JGL

Target: JGL library (173 classes)

Machine: Pentium II, 450 MHz with 128 Meg of RAM

Reading the input and constructing the ORG: 9620 milliseconds (approx. 10 seconds).

Displaying the ORG: 1802 milliseconds.

Computing the firewall: 4267 milliseconds.

Computing the test order for the entire graph: 8115 milliseconds.

Chapter 5 - Polymorphism and Proposed Solutions

Polymorphism allows a runtime reference, called a *client*, to refer to an instance of various classes, called *servers*, to respond to the client's request. Polymorphism means that the same request may be executed differently, depending on the actual server providing the implementation for the request. This mechanism greatly complicates the tracing of dependencies [Corriveau, 96], [Wilde&Huitt, 92]. According to Wilde, "the static analysis will not always be able to identify precisely the dependencies [due to polymorphism] in a program". Therefore, any static analysis that accounts for polymorphism must include all possible dependencies (servers), unless the actual type of the servers can be determined statically [Harrold&Larsen, 96].

We propose two solutions to polymorphism in Java. The first solution, called *polymorphic-bottom substitution*, attempts to collect all dependencies due to the polymorphic inheritance relationship (i.e., the inheritance from an *abstract* class) and the *interface* relationships (i.e., the relationships of a class and its interfaces). We are suggesting the *polymorphic bottom substitution* to (1) identify polymorphic (abstract) classes and interfaces and (2) account for all possible dependencies by determining all possible servers that may be used at runtime. However, there is another other form of polymorphism, the *subtyping* through inheritance of a concrete class, which is difficult to detect via static analysis. In an attempt to cover all dependencies due to polymorphism, we propose a manual refinement technique, the *typed-constraint list* to

handle this form. The typed-constraint list of a variable purposely contains a list of actual types for that variable.

The remaining of this chapter firstly demonstrates that some dependencies are left out in the current class firewall model due to polymorphism. Secondly, it provides an overview of how polymorphism is implemented through dynamic binding. Finally, the proposed solutions to detect those missing dependencies are discussed in detail and other related issues will be raised.

5.1 Demonstration of Dependencies Missing due to Polymorphism in the Current Class Firewall Model

We now demonstrate the missing dependencies triggered by polymorphism on the Bank example. Assume that a Bank object accepts a *withdraw* request from a customer by forwarding the request to the BankAccount object. For some reason, the *withdraw* method implemented in the SavingAccount class is used to carry out the request. Thus, the Bank depends on the SavingAccount class to serve the customer's request. This means that if any change is introduced in the SavingAccount class, it will directly or indirectly affect the Bank class. Unfortunately, the dependency between the Bank class and the SavingAccount class is not readily detectable. The following figure illustrates the dependencies missing due to polymorphism in the Bank example.

Figure 5.1 A sample of the Bank example with dependencies missing due to polymorphism

Because the BankAccount is an abstract class (indicated by the *italic* face), one of its concrete subclasses/descendants must be substituted at runtime. In this example, a concrete subclass/descendant of the BankAccount could be any of the CheckingAccount, SavingAccount, or PreferredCustomerCheckingAccount classes. So the Bank class depends not only on the BankAccount, but also on its descendants. In figure 5-1 above, the big arrows, originated from the Bank node incident into the three descendants of the BankAccount, indicate the missing dependencies triggered by

polymorphism of the abstract `BankAccount` class. It should be noted that the types of dependencies are not identified yet.

5.2 Dynamic Binding in Java

As a strongly typed language, Java requires objects to be typed at compile time. However, it does not require the type to be exact. This means that an object can be typed to its own class/interface or to any class that it inherits/implements from. As a class example, assume that `Rectangle` and `Triangle` are concrete subclasses of the `Polygon` class. Any client of the `Polygon` can refer to an instance of either the `Rectangle` or `Triangle` to carry the specific *drawing* request at any point during runtime. As an interface example, assume now the `Polygon` class implements the `Drawable` interface to ensure that itself and its descendants can respond to the *draw* method. Thus, any client of the `Drawable` interface *must* refer to an instance of any class that implements this interface at run time. In particular, the `Polygon` or `Rectangle`, or `Triangle` must be created in place of the interface `Drawable`.

This implies that the compiler cannot determine whether the server is (1) an instance of the class specified in the type declaration, (2) or an instance of a subclass or descendants, (3) or an instance of a class or its descendants that implements the server interface or server's sub-interfaces. The type of the actual server can only be determined at runtime. This type of binding is called *late binding*, a kind of dynamic binding. It is dynamic in the sense that the actual class creation can be delayed until runtime, but it is

also static since it requires compile-time type checking. Compile-time type checking can be done by the *subtyping* mechanism. Subtyping through inheritance (and interface¹⁰) guarantees that an instance of a subclass/descendant can be substituted into any context in which an instance of the superclass appears without causing any type error because the subclass/descendant has all the common attributes and methods of its superclass. Abstract classes and interfaces exist solely for subtypes because it guarantees (by the compiler) that the polymorphism certainly occurs. On the other hand, for concrete (server) classes, it is hard to determine if polymorphism takes place or not (because of (1) above). Furthermore, inheritance also allows *subclassing* mechanism [see Chapter 2]. In particular, since the subclass uses its own defined external interface, the subclass should not be semantically considered as a subtype of the superclass. This means that polymorphism should not occur among classes belonging to *subclassing* relationship. Unfortunately, Java does not support any mechanism to exclude (or identify) the obsolete inherited interface from a subclass. Concrete (server) classes also fall into this category. Thus, it is impossible to detect whether concrete (server) classes, which are of the form subtyping (including strict-subtyped) or subclassing.

5.3 The *Polymorphic Bottom Substitution* Solution

The proposed *polymorphic bottom substitution* solution detects polymorphic dependencies due to inheritance for subtype based on the *polymorphic* inheritance relationship and the *interface* relationship.

¹⁰ We consider that subtyping through interface is similar to that of polymorphic inheritance.

First let us define some terminology used in this solution. Recall that the polymorphic classes are those declared as abstract classes or interfaces. Polymorphic aggregation relationships are aggregation relationships in which the aggregates are polymorphic classes; whereas, polymorphic association relationships are association relationships in which the associates are polymorphic classes. The labeled edges of an ORG indicate the type of the class relationships. The furthest concrete descendants of a class are those concrete descendants at the furthest distance of a class in each branch rooted by the polymorphic class. The furthest concrete descendants are often the leaves of the tree. Thereafter, for simplicity, we refer to these furthest concrete descendants as leaves. The technique is formalized in terms of ORG as follows:

If a directed edge (A, B) of an ORG is a polymorphic aggregation or association edge, we do the following things:

- Add a new polymorphic aggregation/association edge in the same direction from node A to each of the leaves of the polymorphic class's node B respectively, with labeling AggPol/AssPol respectively, and
- Optionally remove the directed edge (A, B) from the ORG.

Note that we are not concerned with “*abstract*” leaves since there are no substitutions for these classes (no instance of abstract class at runtime). In particular, this solution will instead add polymorphic edges to the closest concrete ancestors of these leave.

JFireClass applies this technique to capture dependencies due to polymorphism as shown in the following figure (5-2).

Figure 5-2 The ORG representation of the Bank Example with the polymorphic bottom substitution

The modified ORG model of the Banking example above adds two new polymorphic association edges from the Bank node to the leaves of the polymorphic BankAccount class node, the PreferredCustomerCheckingAccount and the SavingAccount. The modified ORG also adds two new polymorphic association and other two polymorphic aggregation edges from the Portfolio node to the leaves of the polymorphic BankAccount class node. This modification captures the fact that the firewall of any descendants of the polymorphic superclasses (i.e., BankAccount) would include any

classes that have polymorphic aggregation and association relationships with their polymorphic superclass.

We depict the general presentation of the polymorphic dependencies in the figure 5-3 below:

Figure 5-3 The general ORG accounting for polymorphism

5.4 Proof of the Polymorphic Bottom Substitution

The correctness of our approach to dealing with polymorphism, and its effect on the underlying algorithm of the class firewall model, are established by the following proofs:

1. Does the polymorphic bottom substitution solution cover all possible dependencies due to the identifiable polymorphic forms described above? Yes, since we add new dependencies to the leaves of polymorphic receiver node, guaranteeing that all possible instances of the descendants are covered. The key of the solution is based on the hierarchy of inheritance relationships. Let us assume that a class A depends on a polymorphic class B via the association relationship, and that a class C is an intermediate concrete descendant of B (not a leaf). Let us also assume that this solution fails to include class A into the firewall of the concrete descendant C of class B. This means that, in terms of ORG, there is no directed path that starts from node A to node B through node C via one of the leaves of the rooted node B. This contradicts the hierarchy of inheritance relationships - there must be a path from a leaf of node B to node C by inheritance and also a path from C to the rooted node B. Thus, if there is any change in C, the class firewall can ripple to A via the path $A \rightarrow C$ [details of class firewall computation are discussed in Chapter 3]. In conclusion, the polymorphic bottom substitution solution covers all possible dependencies due to the two identifiable polymorphism forms.
2. Does the polymorphic bottom substitution solution affect the underlying algorithm of the class's firewall? No, since it only introduces additional defined dependencies. No changes should be made to the algorithm to accommodate the technique. But the new outcome obtained from the proposed solution is a superset of the old result.

3. Does the polymorphic bottom substitution solution affect the test order algorithm, especially its feasibility? No, although new cycles may be added, it does not introduce any new relationships that may violate the breaking cycle rule. Thus, the test order algorithm is left intact.

5.5 Revisiting the Bank Example with the Computation of the Class Firewall

The comparison of PreferredCustomerCheckingAccount's firewall before and after polymorphism handled follows:

Figure 5-4 Class firewalls of the PreferredCustomerCheckingAccount before and after polymorphism handled

The class firewall of the PreferredCustomerCheckingAccount class, excluding polymorphism, only consists of itself (the result is shown in the left panel of the figure). However, by considering the dependencies triggered by polymorphism, the class firewall of the PreferredCustomerCheckingAccount class includes the other four classes: the Bank, Bank\$BankEnumerator, Portfolio, and Portfolio\$PortfolioEnumerator classes.

5.6 The Typed-Constraint List Solution

As mentioned in previous section, subtyping through inheritance also allows another form of polymorphism when a superclass is declared as being concrete. Unfortunately, the way of identifying this concrete superclass is identical to that of a superclass for subclassing scheme. The *typed-constraint lists* solution, which we propose here, addresses the “identical” issue. A typed-constraint list is basically a collection of bindings for every variable used in a class. Each binding consists of the variable name and its associated “exact” types that are eventually used at runtime. This proposal solves two problems. First, it helps to pinpoint the ultimate types of variables that are actually substituted at runtime by eliminating improper subclassing “types” and unnecessary polymorphic types at the same time. Second, it can also be used in conjunction with the polymorphic bottom substitution to minimize the numbers of unnecessary polymorphic dependencies. For instance, assume that the Bank class uses a concrete (or abstract) BankAccount instance that requires only the CheckingAccount instance at runtime. In particular, the binding of the BankAccount variable in the typed-constraint list for the Bank class only needs to specify the CheckingAccount. In terms of implementation,

these lists can be either created and updated on-demand by the testers through the supporting tool or imported from other phases of the development process such as the design phase.

In addition, we can minimize the number of possible dependencies due to interface relationships by making use of their strict-data properties. Strictly speaking, classes that implement strict-data interfaces are actually not involved in polymorphism (no common interface). In order to remove unnecessary dependencies by the strict-data interface relationship, we have to make sure that not only the server interface but also all of its superinterfaces are strict-data. In summary, our proposed solutions cover most of the forms of polymorphism that exist in Java.

5.7 Discussion on the Difficulty of Detecting Aggregation Relationships Involving Polymorphic Data Structures (Aggregation Inferencing)

Classes that represent data structures such as HashTable and Vector often contain objects of different types (i.e., all of them descendants of a common type). Such classes are called polymorphic data structures (PDS) [Meyer, 98]. For example, an instance of class Vector in Java can contain any Java class objects (JDK 1.1x). The current class firewall model is able to detect the type of these PDS objects (i.e., Hashtable). However, our concerns here are how to determine the types of the PDS objects' elements and if so, what is the nature of the dependency of the whole class containing the PDS objects with these elements. [Hurdle, 97] states, "Determining the true type[s] of an aggregate based on a [PDS] class proves to be very difficult in a typeless

OOPL...”. This observation also extends to typed languages because it is impossible to determine the type(s) of runtime objects at the compile time. Figure 5-5 below displays a fragment of the Bank class definition:

```
public class Bank{
    .....
    private Vector portfolios = null; // portfolios can store any type of objects
    .....
}
```

Figure 5-5 A fragment in class Bank definition

The class firewall model is able to detect the instance variable “portfolios” of type Vector. But it does not account for the precise type(s) stored in the “portfolios” instance variable. However, we observe that, when the PDS aggregate (i.e., portfolios) is used, the actual type(s) of its elements must be declared in the containing class definition and therefore will be identified. For example, as in our implementation, JFireClass detects the type(s) of elements of the PDS aggregate as association relationships. The following figure shows how the type(s) of the “portfolios” instance variable are detected.

Figure 5-6 An example of how the PDS aggregate is used

To explain this example, the *Bank* class has an instance variable *portfolios* as type *Vector* (see figure 5-5), and the class firewall model has no problem detecting the aggregation relationship between the *Bank* and the *Vector* class (declared type of the *portfolios* variable). However, what we would like to see here is that the model should detect the true type(s) of the *portfolios* variable, which actually is *Portfolios* (since its stored elements of type *Portfolios*). In figure 5-6 above, the *portfolios* variable has been used, and since Java is strongly-typed, the type of its elements must be declared, the type of the parameter *port* is detected as *Portfolio*. The class firewall model indeed establishes association relationships between the class *Bank* and the *Portfolios* (via the *port* variable in method *foo()* or *aPortfolio* variable in method *anotherFoo()*).

However, two problems remain: (1) What if such PDS aggregated objects are not used? and, (2) Is it reasonable to identify such relationships as association relationships? If the PDS aggregated objects are not used anywhere except where they are declared, constructed, or accessed (i.e., accessor, mutator); then indeed there is no coupling between the class and its aggregates. And thus, there is no need to detect such an aggregated object and its elements' types. The following is a sample (figure 5-7) to illustrate the case that there is no dependency between the whole class and the true type of a PDS aggregate.

```

class Whole {
    private Collection aggregate = null;           // where the PDS aggregate declared.
    .....

    public Whole {
        aggregate = new Collection();             // initializing the aggregate
    }
    Collection accessor() {                       // accessing the aggregate
        return aggregate;
    }

    void mutator(Collection col) {
        aggregate = col;                         // setting the aggregate
    }
}

```

Figure 5-7 An example of a class without “true” dependency with its PDS aggregate

The above example reveals that the Whole class definition does not contain any statement involving the manipulation of the PDS elements. Therefore, there is no need to capture the true type of the aggregate.

To answer the second problem we raised, we believe that it is adequate to identify the relationships between a class and its PDS aggregate’s elements as association relationships. First, these elements are associated with their PDS containing class (by the definition of the association relationship) and, by transitivity, these elements are associated with the class containing the PDS aggregates. Second, our goal is to find all possible dependencies.

Let us now turn to the implementation of JFireClass.

Chapter 6 - The Implementation of JFireClass

Figure 6-1 JFireClass 1-1

6.1 Introduction to JFireClass

JFireClass allows testers to build a firewall to identify a set of affected classes by changes introduced in a single class or even a set of classes. This helps testers in conducting selective regression testing on the classes in the firewall as opposed to the entire classes of the program. Furthermore, JFireClass also produces a test order for the class firewall by systematically selecting individual classes to test to further reduce the test effort. The design of JFireClass is based on class firewall model described in Chapter 3. For more details of the class firewall and related information, readers are invited to consult [Kung et. al., 94], [Kung et. al., 95a,b], and [Lueng&White, 92].

JFireClass is written in Visual Age for Java version 1.1, then 2.0. JFireClass can be broken into its four main components:

- (1) The class file parser component. Accepting a set of Java bytecodes as opposed to source code, the parser reads the physical class files and creates the class file objects to represent the physical class files.
- (2) The lightweight reflection component. A set of objects is used to refine the raw information taken from the class file objects. It offers a mechanism for class file object to *introspect* itself to the outside world - to answer all information of the class itself and its members (fields and methods). The component is named as *lightweight* reflection since it offers a subset of Java Reflection API functionality (and other specific requirements for the class firewall model).
- (3) The class firewall construction component. The class firewall toolset provides functions to build the ORG model out of the lightweight reflection objects. The tool set also provides the underlying algorithm to compute the class firewall and the test order. A solution to address polymorphism, namely *polymorphic bottom substitution*, is also in place.
- (4) The graphical user interface component. The interface is composed of several GUI objects such as menus, toolbar, graphical ORG objects, etc., all designed with user-friendliness in mind.

6.2 Advantages of ByteCode Analysis

As mentioned in Chapter 2, Java is both a compiled and interpreted language. Java source code (files with “.java” extension) is compiled to Java platform-neutral code (bytecodes), which is then executed by a Java Virtual Machine (JVM). The bytecodes are stored in separate class files (files with “.class” extension) for each Java class (inner classes are stored separately from their enclosing class) or each Java interface. We decided to analyze Java dependencies using class files instead of the more readable Java source code files for a number of reasons:

Stability/Portability: The Java language is still in the making. This means that there will probably be more language changes in the future. These changes eventually affect the source code syntax. However, the class file format is designed to be immune from these language specification changes. In fact, the class file format is JVM-specific, and since JVM can be implemented by different vendors, it requires a fixed standard format to support Java’s “write once, run everywhere” philosophy. Indeed, with the addition of inner class syntax, the compiler must be modified to adapt for the presence of inner classes in the source code and to correctly convert them to bytecode in the class file. However, there are no changes to the class file format as processed by the JVM. The organization of the resulting class file is specified with enough precision that all 1.1-conforming compilers will produce binary compatible class files.

Efficiency: Class files are structured in a record-like structure (i.e., structure-like in C). So the process of collecting class information is more systematic and indeed more

efficient (i.e., we know exactly where to get what). We don't have to account for every byte in the source code, but only focus on parts in the class files for which we want to collect information. For example, each class file structure has a nested method structure that contains information on all formal parameter types that are used for each method. So instead of reading every byte in a method declaration to extract the parameters' type in the source code, we simply read the method structure to get all needed information in one pass.

Flexibility: Parts of a program may use external libraries (i.e., third-party components, middle-ware, etc.). Changes in external components (i.e., a new version) obviously affect any program that uses them. Thus, dependency analysis tools should consider the presence of such libraries. Unfortunately, they are often not available in source code format. As a matter of fact, any dependency analysis tool that is based on source code limits itself to internal program components. Thanks to JVM specification, dependency analysis based on the class file will inherently include all external Java libraries.

In short, by concentrating on bytecodes as the targeted medium for dependency analysis, JFireClass is made more stable, efficient, and versatile than any source code based tool. In fact, JFireClass will still correctly operate even though changes are made in the Java language. It should be noted that the bytecodes and source code of Java are semantically equivalent, that there is no loss of information. Furthermore, our approach is bytecode specific, and that the source code syntax could be anything as long as it produces the same bytecodes.

6.3 The Class File Parser Component

6.3.1 Class File Structure

In this section, we will show portions of the class file structure that relate to the implementation of JFireClass. For complete information on the class file format, readers are invited to consult [Lindholm&Yellin, 96], [Yourst, 98].

The class files are structured as a record-based structure. They consist of seven distinct parts as the following figure indicates:

Figure 6-2 File class structure

A class file starts with the magic number that identifies the class file format, and versions of the compiler that produced the class file. The constant pool of the class file

(Constant Pool Part) is similar to the symbol table in a conventional language's binary object file. Each item in a constant pool primarily contains types that are referenced by the remaining parts such as the Method Info part and the Interface Info part within the class file. For example, each interface element (i.e., `interfaces[i]`) stored in the Interface Info part contains a valid index of the constant pool, which refers to the interface type. The Class Descriptor part indicates the scope of the class (i.e., `access_flags`), the valid index of the class name in the constant pool and the valid index of its direct superclass name in the constant pool (i.e., `this_class`, `super_class`). The Interface Info part, the Field Info part, and the Method Info part are composed of two items: the size of the array and the array itself. The structures of the arrays are basically similar. Each structure contains specific information. For example, the Method Info part contains all of the information on methods that are defined in the class. The following figure illustrates the structure of the method array (i.e., `methods[]`) in the Method Info part.

Figure 6-3 The Method_info structure and its nested attribute structure

Each method element (i.e., `methods[i]`) contains all of the information belonging to a specific method of the class. The scope of the method can be retrieved from the

access_flags item, the name of the method can be read off in the constant pool table with a valid index retrieved from the *name_index* item. The field descriptor including the formal parameters' types and the method's return type can be read off from the constant pool via the *descriptor_index* item. Other information such as local variable information can be obtained through the *attribute_info* structure item. Different attribute elements can be distinguished by *attribute_name_index* item and each of them may have different structures. JVM provides a set of predefined attributes for method elements such as "Code" attribute. The attribute element may contain nested attributes. For example, the "Code" attribute contains the nested "LineNumber", "LocalVariables" attributes. The following example helps us understand the structure of a method element (the *attributes* item is not involved):

Source Code	Class File	
public int foo(int a, String b, float[] c)	methods[i]	<u>Value at the Constant Pool</u>
	access_flags = 0x1; name_index = 12; descriptor_index = 24;	"foo" (I;L/java/lang/String;(F)I

The value of the *access_flags* item is 0x1, which correctly maps to the "public" scope of the "foo" method. The method descriptor value (i.e., (I;L/java/lang/String;(F)I) needs to be further refined (see Java language specification [SunMicrosystem, 98] for more information). Basically, the method descriptor contains two parts: the formal parameters' type list and the method's return type. The formal parameters' type list is enclosed in parenthesis and each parameter is separated by ";". The return type

immediately follows the right parentheses. The “I” indicates the primitive integer type (i.e., int), while Java classes are identified by the prefix “L” and followed by the full class name (including the subsystem name). An array is specified by “[“ and a nested array level is indicated by the number of “[“. The same interpretation method can be applied to obtain information for the remaining parts of the class file.

The last part of the class file format is the *Added-Attribute* Info part, which applies to the class as a whole. The JVM only processes predefined attributes. However, we can add new attributes in this part without effecting the semantics of the class (or interface) at all. The *Added-Attribute* Info part is very useful for the language to extend its features. The typical example is the “inner classes” in JDK 1.1. The JVM defines an additional “InnerClasses” attribute structure and any inner-class compatible compiler is permitted to put (predefined) “InnerClasses” attribute information to the *Added-Attribute* Info part.

6.3.2 Implementation of the Class File Parser Component

The class file parser component is designed to transform the physical class file into internal (to the tool) Java objects, which we will call *class file objects*. The transformation is a one-to-one mapping between class file format to a class file object. In addition, the class file object contains several other objects representing different parts of the class file structure (i.e., its members). For example, the interface and the method objects are constructed to maintain the information stored in the Interface Info and Method Info parts of a class file respectively. Each of these objects is able to read

the information that pertains to it from the class file. In addition, several helper classes are created to process the input stream in a flexible and efficient manner. The class file parser component is originally written by [Yourst, 98] to demonstrate the class file structure and to extract certain information of class file. We extend this package to extract other information that will assist in determining the association and aggregation relationships.

6.3.2.1 Construction of the Class File Parser

To maintain the class file information, the class file parser component creates a main class file object to represent the class file as the whole and various class file component objects to represent the Field Info part, Method Info part, and so on. Each object is basically defined according to its specification in the JVM. Let us examine a look at the definition of the main class file object – the `JcfClassFile`, and the method and field objects – the `JcfMember`.

```
public class JcfClassFile implements RuntimeConstants
{
    public int hdrMagic;                // valid index of the constant pool
    public short hdrMajor;              // ditto
    public short hdrMinor;              // ditto
    protected String fullClassName;     // extra field here to store actual full class name
    protected JcfConstantPool constants; // constant pool
    public short hdrAccessFlags;        // valid index of the constant pool
    public short hdrThisClassName;      // ditto
    public short hdrSuperClassName;     // ditto
    protected JcfInterfaceCollection interfaces; // collection of interface objects
    protected JcfMemberCollection fields; // collection of field objects (JcfMember)
    protected JcfMemberCollection methods; // collection of method objects (JcfMember)
    protected JcfAttributeCollection attributes; // collection of added attributes – (JcfAttribute)
}
```

Figure 6-4 A snapshot of class file class class definition – the `JcfClassFile`

```

public class JcfMember {
    public short mbrAccessFlags;           // valid index of the constant pool
    public short mbrNameIndex;             // valid index of the constant pool
    public short mbrSignatureIndex;        // valid index of the constant pool
    protected JcfAttributeCollection mbrAttributes; // collection of attributes
                                           // JcfAttribute or its descendants
    protected JcfConstantPool constPool;   // a reference to constant pool for look-up.
}

```

Figure 6-5 A snapshot of the method and field class definition – the JcfMembers

In the original version, the JcfCodeAttribute is defined to maintain the information of the method's code. However, it is not designed to capture any nested attribute information. It simply reads an appropriate block of bytes in the class file and stores it in the JcfCodeAttribute object. We refine the JcfCodeAttribute object to recognize the nested *LocalVariable* attribute within it. The following figure depicts the class definition of the added *LocalVariable* attribute that maps directly to the JVM's predefined *LocalVariable* attribute structure.

```

public class JcfLocalVariablesAttribute extends JcfAttribute
{
    protected short catLocalVariableTableSize;
    protected short catLocalVariableNameIndex[];
    protected short catLocalVariableIndex[];
    protected short catLocalVariableStartPC[];
    protected short catLocalVariableEndPC[];
    protected short catLocalVariableDescriptionIndex[];
    protected JcfCodeAttribute parent;
    protected short catTemp[];
}

```

Figure 6-6 Class definition of JcfLocalVariablesAttribute mapping out the predefined LocalVariable attribute of class file structure

6.3.2.2 Parsing Process of Class File Objects

After the appropriate parsing stream, a main class file object (JcfClassFile) is constructed with streamed class file input (JcfClassInput) to be ready for parsing. The

new `JcfClassFile` object then reads the input systematically and sequentially according to its defined structures. It begins the task by reading the identification parts. When it reads its components that are associated with other objects, it delegates the reading task to those appropriate objects. For example, as it is reading the constant pool, the `JcfClassFile` object will construct the constant pool object (`JcfConstantPool`) and forward the *read* request to `JcfConstantPool` object. In this way, `JcfClassFile` will cause the creation of all other class file component objects. The following figure shows how a `JcfClassfile` object is setup for reading the class file input.

```
public void read(JcfClassInput ist) throws IOException, ClassFormatError
{
    hdrMagic = ist.readIntVerbatim();
    if (!isValidMagic(hdrMagic))
        throw new ClassFormatError("Invalid magic signature");
    hdrMinor = ist.readShort();
    hdrMajor = ist.readShort();
    constants = readConstantPool(ist);    // new JcfConstantPool(ist, this);
    hdrAccessFlags = ist.readShortVerbatim();
    hdrThisClassName = ist.readCPrefer();
    hdrSuperClassName = ist.readCPrefer();
    fullClassName = constants.classNameAt(hdrThisClassName);
    interfaces = readInterfaces(ist);
    fields = readMemberCollection(ist);
    methods = readMemberCollection(ist);
    attributes = readAttributes(ist);
}
```

Figure 6-7 A full implementation of a `JcfClassInput`'s *read* method to read the class file input stream completely

The *readConstantPool()* method simply creates a `JcfConstantPool` object to read itself from the parameterized input stream. The process continues until it reaches the end of the input stream. All of the information from the class file is read and stored in a single `JcfClassFile` object. The following figure (6-8) is an example of how an attribute, the `LocalVariableAttribute`, can read itself from the input stream.

```

public JcfAttribute readAttribute (JcfClassInput ist, short nameIndex, JcfCodeAttribute codeAttr)
{
    if (constPool.utfAt(nameIndex).equals("Code"))
        return new JcfCodeAttribute(ist, nameIndex, constPool);
    else if (constPool.utfAt(nameIndex).equals("LocalVariableTable"))
        return new JcfLocalVariableTable(ist, nameIndex, constPool, codeAttr);
    else return new JcfAttribute(ist, nameIndex, constPool);
}

public void read (JcfClassInput ist)    // in class JcfLocalVariableTable
{
    int code_length = parent.catCode.length;

    for (int i=0; i<catLocalVariableTableSize;i++)
    {
        catLocalVariableStartPC[i]= new short(ist.readShortMax(code_length));
        catLocalVariableEndPC[i]= new short(ist.readShortMax(code_length));
        catLocalVariableNameIndex[i]= new short(ist.readCPref());
        catLocalVariableDescriptionIndex[i]= new short(ist.readCPref());
        catLocalVariableIndex[i]= new short(ist.readShort());
    }
}

```

Figure 6-8 The *readAttribute* method serving as attributes dispatcher – The second method depicting how a particular *local variable attribute* object reads itself from the input stream

The reading process eventually stops as the *Added-Attribute Info* part is completely read. The result of the process is a set of class file objects containing all of the necessary information required for the analysis of class relationships.

6.4 The Lightweight Reflection Component

Once a class file object (JcfClassFile) is created, its information will be redefined and stored in the LWClass object. The LWClass object can determine its structure and present itself in an explanatory way. It reflects the information of a class such as:

1. its class name, super class name, scope;

2. fields' names, types, and constant values if defined; and
3. methods' names, scopes, formal parameters, types of local variables, and its return type.

This component also offers a set of functionalities to aid in analyzing various class relationships. For instance, the LWClass object of a class can return a list of the associates, aggregates for that class. The ORG graph is then built by the resulting information from querying the LWClass objects. In short, the lightweight reflection objects are fully employed by the dependency analysis process.

6.4.1 Construction of Lightweight Reflection Package

Analogous to the Java reflection package, the lightweight class reflection package is built around the Class object, called the LWClass, and its aided components such as LWInterface, LWMethod, LWField to hold refined information of a class file object. During the construction phase of the LWClass, with the JcfClassFile object as its parameter, the LWClass object refines its information pertained to class level; otherwise, it dispatches the task to appropriate components. The LWInterface, LWMethod, and LWField will refine the information pertained to them respectively. The following figure shows the LWClass constructor in which the class information refining process takes place.

```

public LWClass (JcfClassFile jcf)
{
    super( jcf.getConstantPool().classNameAt(jcf.hdrThisClassName),
          jcf.getConstantPool().classNameAt(jcf.hdrThisClassName),
          jcf.hdrAccessFlags );

    superClassName =jcf.getConstantPool().classNameAt(jcf.hdrSuperClassName);
    loadInterfaces(jcf);
    loadFields(jcf);
    loadMethods(jcf);
}

```

Figure 6-9 LWClass, at construction time, loads all needed information for dependency analysis

The first line simply reads the class descriptor of the input class file object and stores the actual data in the LWClass inherited attributes from the LWNameAndType class. The second line put its superclass name into the appropriate attribute. The last three lines are invocation methods to delegate the task to the appropriate components. Let us examine one of those delegating methods, the *loadMethods* method in the figure 6-10 below.

```

private void loadMethods (JcfClassFile jcf ) {

    for (Enumeration e = jcf.getMethods().elements();e.hasMoreElements();)
    {
        JcfMember member = (JcfMember)e.nextElement();
        methods.addElement(new LWMethod(member));
    }
}

```

Figure 6-10 An example of constructing the LWClass object's *methods* attribute

The *LoadMethods* method (1) scans through the class file object's method structure, (2) creates the corresponding LWMethod objects on the fly and (3) stores them into the *methods* attribute of the LWClass. LWMethod objects are actually created in a similar fashion to that of the LWClass. This means that a descriptor of a method will be read

then followed by another method's attributes of interest such as the *LocalVariable* attribute.

6.4.2 Class Information Refining Process

Class file information that is collected from the class file objects is encoded according to JVM specification. There are two major areas that need decoding: (1) the type of variables and (2) their scope/access mode. The types are usually encoded in the class's descriptor, the method's descriptor, and the field's descriptor. The *descriptors* are basically encoded similarly; however, there is some variation among them especially for the method's descriptor. The scope and access modes, hereafter referred to as modifiers, are stored in the *access_flags* attribute of the class, of each method, and of each field. By decoding the class modifier, we can determine whether the current class file input is a *class* or an *interface* and whether it is *public* or *abstract* and so forth. The following sections will explain in detail how JFireClass achieves the class information decoding.

6.4.2.1 The *descriptors* Attribute

The *descriptor* attribute usually contains a type of the attribute. Types are either *primitive* types, which are predefined by the language such as the *boolean* type and numeric types (int, double,...), or are *reference* types, which are ordinary object classes (class type, interface type, array type). JVM encodes those two types differently as in the following figure (6-11):

	Types	Decoded Types
Primitives	Byte	B
	Char	C
	Long	J
	Boolean	Z
		
References	L<classname>;	an instance of the class
	[	single array dimension

Figure 6-11 A sample of types encoding according to JVM specification

For example, the descriptor of an *int* attribute is simply encoded as “I”. The descriptor of a *String* attribute is encoded as “Ljava/lang/String;”. It should be noted that the *fully qualified*¹¹ class name is used. The descriptor of a multidimensional String attribute, such as String str[][][], is encoded as “[[Ljava/lang/String;”.

The descriptor of a method is more interesting. It contains a list of formal parameter types, which are enclosed in parentheses and followed by the method’s return type. See the following example:

Method declaration	Encoded method’s descriptor
String myfoo(int[][] i, Vector v, Panel p)	([[Ljava/lang/Vector;Ljava/awt/Panel;)Ljava/lang/String;

¹¹ A fully qualified class name includes the package name followed by the “/” followed by the subpackage name, if any, followed by the “/”, and so on, followed by the simple class name.

6.4.2.2 Decoding the *Descriptors*

Given a descriptor, LWDescriptor systematically detects if a given input is a primitive, a reference or an array type by examining the first character of the input. If the descriptor is primitive (not 'L' nor '['), the method returns the corresponding type by consulting the primitive look-up table. If the descriptor is a reference type ('L'), the method strips the character 'L' at the beginning and the comma at the end, then returns a fully qualified class name (i.e., java/lang/Vector). If the descriptor is the array type, it strips all '['s and reprocesses as it has done for the first two cases. The *getFullType* pseudo-code below illustrates the process described.

```

public static String getFullType(String type)           // LWDescriptor class
{
    String result = new String(type);                   // make a copy
    if ( isArray(result) ) // first character == '['
        // keep track number of '[' as reading along
    if (isPrimitiveType(result)) // first character == 'I', 'D', 'Z' ect...
        // return primitive type (i.e. int, boolean)
    else
        // return reference type (i.e., java/lang/String)
}

```

Figure 6-12 The *getFullType* pseudo-code in LWDescriptor class

In the case of a method's descriptor, the LWMethod object starts by separating the formal parameter list and the method's return type. The resulting return type will be processed normally. The formal parameter list is processed by reading each type one at a time. Once a type is extracted, it is forwarded to a LWDescriptor object to be processed in a normal fashion.

6.4.2.3 The *access_flags* Attribute

The *access_flags* is a masking value that stores the modifiers used with a class, or methods of the class, or any variable that is used in the entire class. Using a masking technique, the *access_flags* (in a hexadecimal format) can hold more than one piece of information of the class modifiers. For example, if the *access_flags* of a class is 0x0201, the “class” is declared as public (0x0001) and is actually an interface (0x0200). Another example: if the *access_flags* of a method is 0x0009, then the method is declared *public* (0x0001) and *static* (0x0008). The following is the sample of valid modifiers:

<u>Flag Name</u>	<u>Value</u>	<u>Meaning</u>	<u>Used By</u>
ACC_PUBLIC	0x0001	Is public;	Any method
ACC_PRIVATE	0x0002	Is private;	Class/instance method
ACC_PROTECTED	0x0004	Is protected	Class/instance method
ACC_STATIC	0x0008	Is static.	Class/instance method
ACC_FINAL	0x0010	Is final	Class/instance method
ACC_ABSTRACT	0x0400	Is abstract	Any method

Figure 6-13 A sample of modifier's values

6.4.2.4 Decoding the *access_flags* Attribute

The LWModifier class is designed to decode the *access_flags* item. It consists of all static methods providing functionality to determine if the input is matched for a certain modifier. For example, a field object can determine itself whether it is *protected* by delegating the work to the LWModifier class. The following code sample demonstrates this:


```

Class LWField {
    public boolean isProtected(){
        return (LWModifier.isProtected(this.access_flags));
    }
}
Class LWModifier {
    public static boolean isProtected(int access_flags) {
        return ((access_flags&ACC_PROTECTED)==ACC_PROTECTED);}
}

```

Figure 6-14 A sample of codes to show how values in a method's modifier are retrieved

Using such an implementation, our LW-components do not need to maintain a list of “yes/no” (boolean) for each modifier that pertains to them. They can access this information dynamically by making use of the LWModifier class.

6.4.2.5 Determining Class Relationships

Once a LWClass object is constructed, it is ready to analyze its relationships with other classes associated with it. The inheritance, interface, aggregation, and association relationships can be basically obtained by reading the *name* (or type) attribute in its *superclassName*, *interfaces*, *fields*, and *methods* attributes respectively. LWClass class offers an array of methods to acquire those relationships: `getInheritanceRelationship()`, `getInterfaceRelationship()`, `getAggregationRelationship()`, and `getAssociationRelationship()`. These methods basically return collections of respective types that are collected. Note that these collections only store direct relationships. For instance, only the collection returning from `getInheritanceRelationship()` only stores the direct superclass, and there is no need to collect all subclasses and superclasses as in [Hurdle, 97]. The reason for this is that non-directed relationships will be found through the class

firewall algorithm under the transitive dependency relation. The pseudo algorithm for computing the aggregation relationships can be depicted as follows:

```

Class LWClass {
    Collection GetAggregationRelationship(){
        For each element in the fields collection of the class
            Add the LWField.name to the collection
        }
    return the collection    // only interested in the type of the aggregate not the entire LWField
}

```

Figure 6-15 A pseudo-code to return all types of relationships of a class

6.5 The Class Firewall Construction Component

6.5.1 Constructing the ORG from LWClass Objects

The ORG graph plays an important role in the class firewall process. It holds various types of relationships that are captured through LWClass objects. The ORG is a digraph (directed graph) whose nodes are the set of LWClass objects and whose edges represents relationships between class nodes. The edges can be further categorized into six distinct sets: inheritance, aggregation, polymorphic aggregation, association, polymorphic association, and interface edges. The direction of edges shows the dependency. For example, if node A “inherits” node B, then the corresponding inheritance edge’s direction is pointing at node B. The implementation of the ORG is very simple since all information required for its construction is available in the nodes. The following figure (6-16) presents the algorithm to construct the ORG from a set of LWClass objects.

```

Class ORG {
  ORG (LWClass[] classes)
  {
    for each class in the classes collection {
      create a ORGNode and add the class to the node;
      for each relation X = class.getRelationX
        for each class name in the relation {
          if ( the node of the class is not created yet )
            create the node and add it to the ORG graph
          create the corresponding edge between the two nodes
        }
      }
    }
  }
}

```

Figure 6-16 An algorithm to construct the ORG object from the given set of LWClass objects

6.5.2 Computing the class firewalls and test orders

The class firewall and the test order algorithms have already been discussed thoroughly in Chapter 3. Employing the DFS [Cormen et.al., 90] as opposed to the full-blown transitive closure offers the efficiency in deriving the firewall for a class or small set of classes under test. Using DFS to compute the clusters also greatly reduces the complexity of finding test orders. As in our implementation we employ the DFS algorithm with some modifications so that we can exploit information stored in the DFS trees to help us to reduce some work. For example, we don't have to construct extra data structures to represent clusters. Neither do we have to run the entire test order as in [Kung et. al., 95a] in case of an ORG is acyclic. In the following section we will briefly describe the solution to handle polymorphism.

6.5.3 Implementation of the Polymorphic Bottom Substitution

Let us recall the technique as follows. The polymorphic bottom substitution solution simply finds all polymorphic aggregation and association edges of the ORG, optionally removes them, and adds new polymorphic aggregation and association edges respectively to the leaves of the receiver polymorphic class's nodes. The implementation of the technique requires an added function to the ORG object to retrieve the descendants' information. The algorithm can be summarized as follows: as the process hits a polymorphic aggregation edge or association edge, it will ask for a set of leaves rooted at the polymorphic node and sequentially add a new respective edge to each of member of the set. Consider the pseudo-algorithm that follows:

```

Class ORG {
    void addPolymorphicEdges(){
        for each node (A) in the ORG graph
            for each aggregation edge/association edge (A, B)
                if (B is interface or abstract)
                    add the set of leaves of B to the set of polymorphic edge set of A
    }
}

```

Figure 6-17 The implementation of the bottom forwarding solution to polymorphism

Interestingly enough, to compute the set of leaves for a (polymorphic) node, we employ the DFS algorithm. The idea is simple. The DFS_Visit recursively traverses along the incident inheritance and interface relationship edges until a leaf node is visited. The leaf node will pass its ID to its parent, by adding its name to the set of leaves of its parents (plural in case of the interface inheritance relationships). Having been backtracked and explored all accessible descendants, the intermediate node contains all information of its leaves and transfers all the information to its parents and so on. The process eventually

stops when all nodes that are reachable from the originally given node are explored. Let us examine the DFS_Visit_ComputeLeaves to compute the set of leaves for a node as follows:

```

Class ORGraph {
void DFS_Visit_ComputeLeaves(ORGNode A)
{
    A.color = DFS.GRAY;

    for each incident inheritance or interface edge (A,B) of the node A{
        if (B.color == DFS.WHITE ) {
            B.parent = A
            DFS_Visit_SetLeaves(B);
        }
        else if ( B.color == DFS.BLACK) // interface may have multiple parents
            A.addLeaves(B.getLeaves());           (*)
    }

    if ( A.isLeaf)
        A.parent.addLeaves(A.getName());          (**)
    else
        A.parent.addLeaves(A.getLeaves())          (***)

    A.color = DFS.BLACK;
}

```

Figure 6-18 The pseudo algorithm to compute leaves for a polymorphic class node

In terms of the algorithm, the “for loop” basically carries out the graph traversal along the inheritance and interface paths emanating from the given node. If a node is a leaf (**), it adds its name to the set of leaves of its parent. Otherwise, it knows itself as an intermediate node and adds its computed set of leaves to that of its parent (***). It marks itself *visited* and the process backtracks to its parent node. The leaves-computation will propagate all the way up the tree rooted at the given node. Note that in the case that a child has already been visited from other (interface) parent (*), the current parent must add the already-computed set of leaves of its child to its set of leaves.

6.5.4 Implementation of the Typed-Constraint List

A typed-constraint list of a class is of a collection of bindings for each variable used in that class. A binding consists of the variable name and its “exact” types that are eventually in use at runtime. For instance, if the “bankAccount” aggregate of the Bank class only allows instances of CheckingAccount to be substituted, the collection of bindings for this aggregate should only contains a single binding (bankAccount, CheckingAccount). The following figure suggests a format of the typed constraint list:

<u>Variable name</u> – (List of types)
bankAccount - (CheckingAccount)
customer- (Customer, PreferredCustomer)

Figure 6-19 A sample of a type-constraint list for the Bank class

In our implementation, upon request of building the typed-constraint list for a certain node’s class, a dialog will pop up with a list of all variables defined within the class and the corresponding lists of all possible types for each variable. Possible types of a variable are either all descendants of the variable’s class or all descendants of all classes that implement the variable’s interface. The dialog allows testers to build “exact” types of a variable from a computed list of all possible polymorphic types associated with that variable. This will enforce the correctness of the tool. The gist of this implementation lies in the computation of the all possible types (or descendants) for a certain class. The algorithm, we propose here, is very similar to the *polymorphic bottom substitution* algorithm. In particular, instead of collecting only information of the leaves of the inheritance hierarchical tree, this algorithm also gathers information of all intermediate node’s classes. An intermediate node must pass information of its leaves and itself to its

parents. This ensures that a class obtains a set of all of its descendants. Once the information is confirmed, the ORG automatically updates itself.

6.6 Friendly GUI Components with Graphical ORG Representation

In this section, we will show you how to use JFireClass to conduct the class firewall and test order computations. We will also introduce some other interesting aspects of the tool as we go along. Basically, JFireClass provides a rich and intuitive set of GUI control elements. It is composed of a menu, a tool bar, and a popup-menu component to allow testers to interact with the system. A scrolling canvas panel serves to display the outcomes. Testers simply conduct tasks via the GUI components and the result is shown in the canvas panel. The ORG contained in the panel is also available for interaction with testers. Testers can arrange the ORG for a better layout - positioning ORG node and edges, or selecting the CUT, and so on. The following figure (6-20) illustrates the main GUI components of JFireClass.

Figure 6-20 The JFireClass's GUI components

6.6.1 Supplying the Class File Input

The first thing testers must do is to supply the input, which is basically the set of Java class files (with “.class” extension). Rather than choosing one class file at the time, JFireClass accepts the entire class files from a file containing all class file names including their directory and path. The following figure 6-21 illustrates the process of getting the input.

Figure 6-21 The Custom FileDialog aids in the selecting input process

To select input, say “car.lst”, testers must click on the “Load” button or “Load” menu item from the menu bar or from the popup menu bar (right-mouse click to get the popup menu). Once the input is selected, the validation of file types is undertaken. If the wrong type is supplied, the process simply terminates. To select the input, testers must click on the “Load” item from the menu bar or from the popup (right mouse click to show the popup). Once the input is selected, the validation of the file type is undertaken. If the

wrong type is supplied, the process simply terminates. Otherwise, the process continues as the following figure (6-22) depicts.

Figure. 6-22 Transforming class files into LWClass objects.

Upon receiving the class file input from testers, JFireClass immediately carries out the class file parsing and the refining processes.

6.6.2 Displaying the ORG

As the input process completes, JFireClass is ready to display the ORG. To do so, testers need only to click on the proper GUI component (i.e., the ones caption with “ORG” or “Display ORG” respectively). The following figure 6-23 shows the ORG of the Car example generated from JFireClass.

Figure 6-23 Displaying the ORG of the Car example

To have a better layout for the ORG, testers may select the nodes (by clicking on them) and drag them to the desired locations. The edges are dynamically re-arranged accordingly. JFireClass is also capable of displaying the ORG in UML short form as discussed in Chapter 4. To do so, testers can select “Option→UML mode” from the menu bar or “UML mode” from the popup menu. And toggling the “UML mode” will bring the ORG back to the traditional look.

Figure 6-24 Displaying the ORG of the Car example in the UML short form

6.6.3 Computing the Class Firewall for a CUT

First, testers must specify the CUT node by double clicking on the desired node, say the IRun node. The CUT node becomes red (it is seen dark gray in the figure). As in the above process, testers need only to click on the appropriate “Firewall” button or “Compute Firewall” menu item under the “ClassFire” menu bar to conduct the firewall computation for the CUT node. The affected nodes are “grayed out” (they are light gray in the figure). Figure 6-25 depicts the firewall computation of the single “IRun” node.

Figure 6-25 The Class Firewall for a single IRun node

To select more than one node for the CUT, testers must hold the “Ctrl” key while clicking on the other ones. The set of selected nodes will turn red to indicate they are the CUTs and the affected others will turn back to normal (no longer gray-out) to be

ready for another class firewall computation. The following figure 6-26 shows the class firewall of IRun and Engine nodes.

Figure 6-26 The Class Firewall for a set {IRun, Engine}

6.6.4 Computing the Test Order for a CUT and for the Entire System

Once the firewall computation is complete, we are ready for the test order. With a simple click on the “Test Order” button or on the corresponding menu item, the test order will be displayed immediately. Testers can either compute the test order for the firewall only or for the entire system by selecting “ClassFire → Compute TestOrder → FireWall Only” or “ClassFire → Compute TestOrder → Entire Graph” respectively on either the menu bar or the popup menu. Two figures below 6-27 and 6-28 show the test

The test order for the firewall of the CUTs is (IRun, Vehicle, Engine, Car, Person) and the test order for the entire system is (IRun, Vehicle, Engine, Tire, Car, Person).

6.6.5 Activating the Polymorphism Feature

For purposes of demonstration, JFireClass does not account for the polymorphism by default. To activate it, testers need to toggle “Option→Account for Polymorphism” from the menu bar or from the popup menu. The ORG will be automatically redrawn to reflect the new polymorphic dependencies as seen in the Bank example in Chapter 4.

6.6.6 Building “Exact” Polymorphic Types Using the Typed-Constraint List

To use the typed-constraint list, testers need to select the desire class node and click on either the “T-C List” button of the tool bar or the “Typed-Constraint List” item of the popup menu bar. The dialog will be shown up as figure (6-29) below.

Figure 6-29 The typed-constraint list for the Portfolio class in the Bank example

The dialog consists of three panels. The left-most panel displays all variables used in the selected node's class. A variable is an aggregate if the text confined within the "[]" is shown as "Field". Otherwise, the variable is an associate and the text implies the method that defines the associate. For example, in the figure 6-29 above, the "accounts" is an aggregate; whereas, the "acct" is an associate defined in the *addNewAccount* method. The right-most panel displays all corresponding subclasses/descendants of the selected variable's type in the left-most panel. The middle panel shows the polymorphic type list built by testers purposely contains only types that are eventually used. The polymorphic type list is defaulted to the set of leaves generated from the *polymorphic bottom substitution* computation if the variable's class is declared as an abstract class or an interface. Otherwise, the polymorphic type list will be empty since we don't know if there is any polymorphic dependency. To add a type to the polymorphic type list, testers only need to select the desired item from the right-most panel and move it to the middle panel by clicking on the "Add" button as shown in Figure (6-29) above. Conversely, to remove a type from the polymorphic type list, testers need to select the desire item from the list and click on the "Delete" button. Clicking on the "OK" button confirms the build and updates the ORG accordingly; "Cancel", cancels the entire process. Consider the following figure (6-30).

Figure 6-30 Updating the ORG accordingly to the typed-constraint list of the Portfolios class

The figure (6-30) shows that the polymorphic edges connecting the Portfolio node to the PreferredCustomerCheckingAccount node and the SavingAccount node have been replaced by those connected to the CheckingAccount node as the result of refining the typed-constraint list for variables declared as the BankAccount type in the Portfolio class. It is notable that the polymorphic edges from the Bank node to the leaves of the BankAccount node still remain unchanged.

Chapter 7 - Conclusion and Remarks

Throughout this thesis we have emphasized the importance of selective regression testing for the software maintenance phase. Impact analysis made selective regression possible by identifying the affected parts due to a change made to the system so that we can focus on the affected parts only as opposed to the entire system. Our tool, JFireClass, is designed to support dependency impact analysis in Java, and therefore is appropriate for selective regression testing. We have also discussed problems introduced by polymorphism and have provided solutions. This chapter summarizes the various contributions of this thesis. Future works for JFireClass will also be considered.

7.1 Summary of Contributions

In this paper, dependency analysis in Java has been explored. A new relationship, the interface relationship, has been added to the set of dependencies found in [Kung et. al., 95a]. Several refinements have also been added within the four established relationships - the polymorphic aggregation, the polymorphic association, and the implicit inner-class aggregation relationships, to name a few. These new relationships allow us to capture more dependencies. We have also analyzed the effects of these new relationships on regression testing and provided “proofs” to support our findings.

The class firewall model has also been improved – the class firewall and the test order computations. As a result, we have indeed proposed more efficient and simpler

algorithms. We have used the Strongly-Connected Component algorithm [Cormen et. al., 90] to compute the clusters for deriving the test order. The test order's performance is significantly improved. We have also considered when the class firewall computation should be performed. We have investigated the DFS [Cormen et. al., 90], to determine the class firewall on demand with an incremental update fashion rather than set up the entire dependencies using a full-blown transitive closure of the ORG up front.

We have considered polymorphism and presented the polymorphic bottom substitution solution and the typed-constraint list solution to handle it. The polymorphic bottom substitution technique covers all possible *polymorphic* dependencies via the polymorphic inheritance and interface relationships; whereas, the type-constraint list complements unidentifiable *polymorphic* dependencies. Blending these two solutions together, we have offered a *thorough* dependency analysis in the sense that all possible dependencies are covered by one of these solutions. We ideally should achieve a *precise* dependency analysis by making full use of the typed-constraint list solution in our tool. We have also determined that *aggregation inferencing* is automatically handled through the Java class relationship identifying process.

Finally, we have proposed the JFireClass tool that implements the concepts presented in this thesis. JFireClass constructs the ORG presentation out of the Java bytecodes as opposed to the source code. We have determined that the Java bytecode is more useful for analysis based on its stability, portability, and efficiency. With bytecodes, we (i)

extend our tool's ability to analyze external libraries, which is often desirable (since it is part of the entire target system) but generally impossible through source code analysis. We also (ii) offer significant improvements on the time-consuming I/O process of parsing the source code input by systematically reading the compact, record-like bytecode format. We can confidently make use of the tool though changes might occur in the Java language. Indeed, we have observed that the bytecode is insulated from syntax changes. JFireClass offers a rich GUI environment where testers can directly interact with a graphical ORG in two different modes: the traditional class firewall model and the prevailing UML model. JFireClass computes the ORG based on the four major relationships along with their refinements, and also computes the test order upon the testers' interaction with the ORG (GUI) component. By taking into account the advantages of using bytecode, by making use of efficient algorithms, by demonstrating the tool on a commercial Java library, and by handling problems stemming from polymorphism, JFireClass proves itself as a workable OO impact analysis tool, one that brings class firewall theory to reality.

7.2 Future Work

Software often comes with a "wish list" of future enhancements. JFireClass is not an exception. In fact, there are several areas open for future development. We list them in decreasing order of priority.

7.2.1 Possible Formation of Other Types of Dependency Resulting from Concurrency (and Distribution)

Unlike other programming languages, where concurrency facilities are supplied by the target platform and operation systems, Java offers concurrency as a part of the language. Concurrency processing refers to the situation where more than one process runs simultaneously within a program. These processes may be executed independently of one another but may also cooperate together to complete a task. Threads/processes cooperate by communicating with others via shared variables or/and message passing. As they cooperate in either way, a *concurrent dependency* comes in play. We believe that there are at least three types of couplings in a concurrent context. Readers are invited to consult [de Champeaux, 93] for a more in-depth understanding about concurrent couplings.

Data coupling

As threaded objects attempt to access a concurrent shared resource (container), the container should be able to coordinate these threads' activities to ensure the consistency of its state. In other words, these threads depend on the others via the container's synchronization.

Protocol coupling

An action takes place when a message that is being sent from a sender to a receiver executes. This type of message is known as send-and-forget (i.e., asynchronous) where the sender does not need to wait for the receiver's receipt upon its

completion. However, in concurrent settings, a sender and receiver should be able to communicate at more complex levels. This kind of communication establishes a dependency between these independent objects.

Identity coupling

An object such as a mediator, a scheduler, or a network server, is capable of managing a set of objects to accomplish its tasks. For instance, a network host accepts requests from clients to connect them to a certain port of the server. Due to certain constraints of the network at runtime, the network host object may use different instances, which are under the same abstraction, to carry out the service. These objects, usually threaded objects, may not themselves depend on each other in general but under this circumstance they are regulated by another object.

We think that these dependencies are dynamic; thus, it is difficult to analyze their impact through a static analysis process. The task of dependency identification would involve identifying the potential threaded objects (or processes) and concurrent environments (containers) where the concurrent objects are cooperating. The coupling of these dependencies would have to be determined in order to support the cycle breaking rule.

7.2.2 Cluster Testing

Java provides an excellent way to group logically related classes together through the *package* mechanism. A system can be structured into *packages*, each of which may

further contain sub-packages and so on. For example, JFireClass is comprised of four packages and several sub-packages within them. Classes within a package are often logically related and classes within sub-packages are even more strongly related. We believe that, due to the definition of package, cluster testing could be done by promoting packages (or subpackages) as units for testing. We foresee problems in adapting the class firewall theory at the cluster level in terms of transitivity dependency. [Fowler&Scott, 97] provides a definition of dependency of packages as “dependency between two packages exists if any dependency exists between any classes in the packages”. These authors also stress that “[h]owever, with packages, the dependencies are not transitive. A transitive dependency makes it difficult to limit the scope of changes with compilation”.

7.2.3 Dynamic Loading in Java

Java implements dynamic loading. Dynamic loading in Java refers to a class that is not linked-in at compile time, but loaded at runtime when it is needed. We have not thoroughly investigated the impact of this mechanism. In Java, a class can be loaded dynamically by its *name*. Once the target class is loaded on the fly, the object of the class can be instantiated for use. See the following example:

```
Class aClass = Class.forName("TargetClass"); //Load the TargetClass class dynamically
```

To explain this example, the variable *aClass* is assigned to a dynamically loaded TargetClass class by invoking the *Class.forName()* method whose parameter is of the

desired type. In our current implementation, the class firewall model is only able to detect the variable *aClass* of type *Class* (instead of *TargetClass* type). The problem lies in the Java implementation of dynamic loading. At compile time, Java compilers will ignore the type checking process and classes/dependencies resolution for such a dynamic loading mechanism. Consequently, the bytecode does not recognize “*TargetClass*” as a class. The class will eventually be loaded into the Java runtime environment whenever the line containing the dynamic loading’s declaration is fetched for execution via the Java *ClassLoader* object. In short, under this current implementation, it is possible that certain relationships through dynamic loading may be missed. Further study is required to come up with a concrete approach to tackle the dynamic loading effect. It should be noted that Java vendors might have different implementations for dynamic class loading.

7.3 Final Remarks

The ability to correctly, efficiently and comprehensively test an OO system has long been a Holy Grail of sorts for developers. To many, the testing activity represents a cumbersome, time-consuming and frequently complex process, lacking in easy-to-use and capable automated tools. One of the more difficult areas of testing involves determining the effect of changes and the subsequent testing sequence strategy. These issues are addressed by *JFireClass*.

JFireClass implements and improves upon some of the most recent advances in impact analysis. It provides useful features such as the ability to interact with an *ORG*

representation of a Java system with the purpose of measuring the impact of changes to the system, allowing the tester to easily play “what-if” scenarios. It can also suggest the most effective approach to testing affected components. In short, JFireClass is an effective and easy-to-use product, essential for any Java testing environment.

Bibliography

[Aho et. al., 83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, "Data Structures and Algorithms", Addison-Wesley, Reading, MA, 1983.

[Benhajla, 96] Halim Benhajla, "Traceability in Object-Oriented Quality Engineering", Master Thesis, University of Ottawa 1996.

[Binder, 96] Robert Binder, "Testing Object-Oriented Software – A survey Software Testing, *Verification, and Reliability*," vol. 6, 1996, pp. 125-252.

[Bohner, 91] S. A. Bohner, "Software Change Impact Analysis for Design Evolution," proceedings, IEEE Computer Society Press, Los Alamitos, California, 1996, p.p. 67-81.

[Bohner&Arnold, 96] S. A. Bohner, R.S. Arnold, "An introduction to Software Change Impact Analysis," proceedings, IEEE Computer Society Press, Los Alamitos, California, 1996, pp. 1-26.

[Booch, 94] Grady Booch, "Object-Oriented Analysis and Design with Applications", 2nd Edition, Addison-Wesley, 1994, 0805353402

[Booch et. al., 98] Grady Booch, James Rumbaugh, and Ivar Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1998, 0201571684.

[Coad&Yourdon, 91] P. Coad and E. Yourdon, "Object-Oriented Analysis", 2nd Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Cormen et.al., 91] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms", 4th edn., The MIT Press, Cambridge, Massachusetts London, England, pp. 477-497.

[Corriveau, 96] J.P. Corriveau, "Traceability and Process for Large OO Projects," *TOOLS-USA '96*, August 1996, Santa Barbara, California, U.S.A.

[de Champeaux et.al., 93] Champeaux, D., Douglas Lea, and Penelope Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.

[Firesmith, 93] Firesmith, D. G., "Testing Object Oriented Software," Proceedings of the Eleventh International Conference on Technology of Object-Oriented Languages and Systems, 1993, Prentice Hall, Englewood Cliffs, New Jersey, USA., pp. 407-426.

[Flanagan, 97] David Flanagan, "Java In A Nutshell", 2nd edn., Chapter 5, O'Reilly, May 1997.

[Fowler&Scott, 97] Martin Fowler and Kendall Scott, UML Distilled “Applying the Standard Object Modeling Language”, Chapter 7, p.p. 114, Addison-Wesley Longman Inc., 1997.

[Jacobson et. al., 92] I. Jacobson, M. Christerson, M. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering A Use Case Driven Approach*, Addison-Wesley, Reading, Massachusetts, U.S.A.

[Grant, 97] D. Grant, *Java Language Reference*, The Java Series™, O'Reilly & Associates, 1997.

[Harrold&Soffa, 89] Harold, M.J., and M.L. Soffa, “An incremental approach to unit testing during maintenance,” proceedings, IEEE Computer Society Press, New York, 88, pp. 362-367.

[Harrold et. al., 93] M. J. Harrold, IEEE, and Brian Malloy, “A unified Interprocedural Program Representation for a Maintenance Environment,” proceedings, IEEE Computer Society Press, Los Alamitos, California, 1996, p.p. 262-271.

[Harrold&Larsen, 96] M.J. Harrold, L. Larsen, “Slicing Object-Oriented Software,” 18th International Conference on Software Engineering, pg. 495-505, Mar. 1996.

[Harrold&Liang, 98] M.J. Harrold, D. Liang, “Slicing Objects Using System Dependence Graphs,” Technical Report The Ohio State University.

[Horwitz et. al, 90] Susan Horwitz, Thomas Repts, and David Binkley, “Interprocedural Slicing Using Dependence Graphs,” proceedings, IEEE Computer Society Press, Los Alamitos, California, 1996, p.p. 137-171.

[Hurdle, 97] J. Hurdle, “Introducing Fireworks: A tool for Regression Testing within the Smalltalk Environment”, Master Thesis, Carleton University 1997.

[Kung et. al., 95a] D. Kung, J. Gao, P. Hsia, P. Lin, and Y. Toyoshima. Chen, “Class firewall, test order, and regression testing of object-oriented programs.” *Journal of Object Oriented Programming*, Vol. 8., No. 2, 1995, pp. 51-65.

[Kung et. al., 95b] D. Kung, J. Gao, P.Hsia, Y Song, Y. Toyoshima, and C. Chen, “Developing an Object-Oriented Software Testing and Maintenance Environment,” *Communications of the ACM*, Vol. 38, No.10, 1995, pp. 75-86.

[Kung et. al., 94] D. Kung, J. Gao, P.Hsia, F. Wen, Y. Toyoshima, and C. Chen, “Change Impact Identification in Object Oriented Software Maintenance.” Proceedings of the International Software Maintenance, pp. 202-211, 1994

[Lejter et.al., 92] M. Lejter, J. Meyers, and S.P. Reiss, "Support for maintaining object-oriented programs," *IEEE Transactions on Software Engineering*, Dec 1992.

[Leung&White, 92] Leung, H.K.N. and Lee White, "A Firewall Concept for both Control-flow and Data-flow in Regression Integration Testing," *Proceedings of Conference on Software Maintenance*, Orlando, Florida, 1992, pp. 262-271.

[Leung&White, 91] Leung, H.K.N and Lee White, "A Cost Model to Compare regression Test Strategy," *Proceedings of Conference on Software Maintenance*, Orlando, Florida, 1992, pp. 262-271.

[McGregor&Korson, 94] J. McGregor, and T.D. Korson, "Integrating Object-Oriented Testing and Development Process," *Communication of the ACM*, vol. 37, no. 9, pp. 59-77, Sept. 1994.

[Meyers, 79] Myers, G.J., "The Art of Software Testing," Wiley, New York, U.S.A.

[Meyer, 97] Bertrand Meyer, "Object-Oriented Software Construction," Prentice Hall, Helmet Hempstead 1997.

[Moreton, 90] Robert Moreton, "A Process Model for Software Maintenance," *Journal Information Technology*, Vol. 5, 1990, pp. 100-104.

[Moser, 90] Louise E. Moser, "Data Dependency Graphs for Ada Programs," *proceedings, IEEE Computer Society Press*, Los Alamitos, California, 1996, p.p. 172-183.

[Perry&Kaiser, 90] D.E. Perry and G.E. Kaiser, "Adequate Testing and Object-Oriented Programming," *JOOP Vol3, No.2*, January 1990, pp. 13-19.

[Rothermel&Harrold, 94a] G. Rothermel, M. J. Harrold, "Selecting Regression Tests for Object-Oriented- Software," *Technical Report TR-94-111*, Clemson University, March 1994.

[Rothermel&Harrold, 94b] G. Rothermel, M. J. Harrold, "A Framework for Evaluating Regression Testing Selection Techniques," *Proceedings of the 16th International Conference on Software Engineering*, pp. 201-210, May 1994.

[Salus, 98] Peter H. Salus, "Object-Oriented Programming Languages", *Handbook of Programming Languages*, Macmillan Technical Publishing, Vol. 1, 1998, pp. 109-110.

[Sharon, 96] David Sharon, "Meeting the Challenge of Software Maintenance," *IEEE Trans. Software Eng.*, Vol. 16, No 5, Jan. 1996, pp. 122-125.

[Stevens et. al., 74] W. Stevens, G. Meyers, and L. Constantine, "Structured Design", IBM System J., Vol. 13, No. 2, 1974.

[Lindholm&Yellin, 96] Tim Lindholm, Frank Yellin, "The Java™ Virtual Machine Specification", The Java series, Addison-Wesley, 1996, available via <http://www.java.com/docs/vmspec/2nd-edition/html/VMSpecTOC.doc.html>".

[Turner&Robson, 93] C.D. Turner and D.J. Robson, "State based Testing and Inheritance," *Technical Report: TR 1/93 University of Durham England, 1993*

[Wilde&Huitt, 92] Norman Wilde and Ross Huitt, "Maintenance Support for Object-Oriented Programs," IEEE Transactions on Software Engineering, Vol. 18, No. 12 December 1992, pp. 1038-1044.

[White, 91] L.J. White, "Software testing and Verification." Elsevier Science publisher B.V. 1991.

[Yourdon, 78] E. Yourdon, "Structured Walkthroughs", Second Edition, Yourdon Press, New York, 1978.

[Yourst, 98] Matt T. Yourst, "Inside Java Class Files," Dr Dobb's Journal, January 1998, pp. 46-52.