Identifying Modifications and Generating Dependency Graphs for Impact Analysis in a Legacy Environment

Identifying Modifications and Generating Dependency Graphs for Impact Analysis in a Legacy Environment

By
Asif Iqbal, B.Sc. Engg.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Applied Science
Department of Computing and Software
McMaster University

Master of Applied Science (2011)
(Computing and Software)

McMaster University
Hamilton, Ontario, Canada

TITLE:	Identifying Modifications and Generating Dependency Graphs for Impact Analysis in a Legacy Environment
AUTHOR:	Asif Iqbal B.Sc. Engg., (Computer Science and Engineering) Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
SUPERVISORS:	Dr. Alan Wassyng and Dr. Mark Lawford
NUMBER OF PAGES:	xviii, 150

To my mom and dad

Abstract

Large enterprise level systems often have their own application software layer wrapped over large software tools or products from commercial vendors. From time to time, vendors release patches which update or enhance the products to meet various requirements. However, applying the patches often introduces risk that the wrapper software layer might behave incorrectly, especially if the customer has little knowledge of the linkage between the application layer and the vendor provided system (for example, because the application itself is a legacy system). So there is always the need for analyzing the impact of patches and reducing the risk in applying them. Impact analysis depends on two sources of knowledge – the physical modifications made by a patch and a dependency graph of the entities in the system. This thesis provides an empirical approach to finding modifications and generating dependency graphs that can be used for impact analysis.

The work presented here is actually part of a large reverse engineering project, which deals with a huge system consisting of a legacy customer application layer, Oracle E-Business Suite and Oracle database. To reduce the risk introduced by Oracle patches, the customer currently executes all tests of their current test suite, which is extremely expensive in terms of money and time, with the added drawback that the testing that is performed is not targeted in any way at the entities that have been changed or that depend on entities that have been changed. The ultimate goal of the project is to provide the customer with a reduced and focused test suite after analyzing the impact of patches. Although a lot of work has been carried out to date regarding impact analysis, program dependency graph generation and regression test selection, none of them deals with such a huge domain as involved in the E-Business

Suite. Moreover, none of them was applied to a system where one component is a legacy layer.

We restrict ourselves only to Java class (bytecode) files for the sake of this thesis. We use static analysis on XML representation of class files produced by off-the-shelf bytecode analysis tools. For finding modification between two successive versions of a class file, we compare the two XML representations and generate the output in XML format. For generating a dependency graph among the entities in the Java environment, we analyze all the XML files, extract dependency information using an empirical technique which we call access dependency analysis, and finally store all this information in a database for use in the impact analysis process. Both of these processes are fully automated without any human intervention.

Acknowledgements

This thesis would not have been possible without the support of many people. Many thanks to my supervisors, Alan Wassyng and Mark Lawford who have always been very flexible to me and have guided me throughout the whole research. I cannot help thanking specially Alan Wassyng for suggesting modifications and correcting the language. Also, special thanks to Wolfram Kahl whose witty suggestions helped me tackling problems in critical moments. Thanks also to Tom Maibaum who offered guidance and support. Another name I should mention is Chris George who we had weekly meetings with on the project (on which this thesis is based on) and received valuable advice and guidance, not to mention the help and support I got from my colleagues Akbar Abdrakhmanov and Wen Chen. Finally, thanks to my parents and numerous friends who endured this long process with me, always offering support and love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Large business organizations spend a huge portion of their resources and budget in maintaining existing software, upgrading it with new features and adapting it to new environments and platforms. A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions [Pig97]. This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system. In the software lifecycle, in addition to bug fixing, maintenance is inevitable for reasons like new customer or business requirements or introduction of new platforms etc. besides fixing bugs [Cho05]. Maintenance can be defined as the set of activities that occur after the software has been deployed [GJM03]. A substitute for maintenance may be the development of completely new software from scratch. However, that is usually quite impractical and may be even infeasible considering the limitations of time, resources and budget of the organization; keeping in mind that companies make large investments in developing software, creating infrastructure and organizational practices around the software and in training users [Cho05]. So, existing software applications are always important assets for these organizations, and they need to be maintained over the course of time.

However, many of the existing systems were developed years ago, some

written in older languages while some others are no longer maintained by the developers who originally developed those. These legacy software systems are therefore extremely difficult to maintain and modify. While most software engineering approaches focus on the *forward engineering* – that is, the forward development process in which we move from initial requirements to logical design and from design to implementation, this thesis, instead, analyzes the implemented software system and extracts useful information from it to aid our customers in maintaining their systems.

With the help of extracted information and the continuous process of maintenance, in the course of time, it may be possible to re-engineer the whole system. Re-engineering is the process of through which an existing system undergoes an alteration, to be reconstituted in a new form [GJM03]. The re-engineering process generally consists of two phases. In the first phase, the existing software system is analyzed to extract useful information, patterns, dependency among components and high level design. which will aid understanding how the system works. This is usually called *reverse engineering.* In the second phase, the information extracted in the reverse-engineering phase is used to build up the whole system or parts of the system from scratch. For the second phase to be carried out successfully, the reverse engineering phase needs to be able to extract information in a comprehensive manner. Complete documentation consistent with implementation aids the reverse engineering process to a great extent. Unfortunately, in most of the legacy systems, complete documentation is not available. So the software engineer often needs to go through the tiresome process of recovering the design from the code and rebuilding the design from low level code [Cho05]. Lack of proper documentation often turns out to be be one of the main factors affecting the cost and efforts of reverse engineering. To make matters worse, during the development of many legacy software systems, documents were not properly updated during maintenance of the software; leaving them in an inconsistent state with respect to the implementation [Cho05].

Our project is a bit different. We have a large system with the customer application layer sitting upon the Oracle E-Business Suite and Oracle database (**Figure 1.1**). Patches released from Oracle corporation are periodically ap-

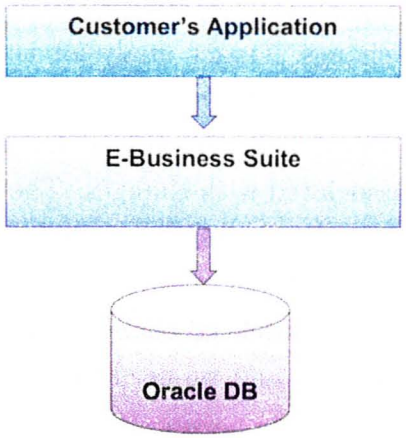


Figure 1.1: Customer's System

plied to enhance or upgrade the Oracle products, i.e., the E-Business Suite and the database. However, there is always the possibility of some change in behaviour in the customer application layer there after the application of a patch. We assume that the customer (who owns/developed the customer application) has a huge test suite consisting of thousands of tests and the customer executes all those tests each time a patch is applied for finding the potential changed behaviour. Obviously, executing all the tests is extremely expensive in terms of both money and time. Companies spend weeks and millions of dollars in executing those tests. So they are in need of a better way of determining which functions in the customer application may be affected by the changes to the Oracle system. The problem is made even more difficult by the lack of proper documentation of the legacy application. The reverse engineering project (what we call the impact analysis project) undertaken at McMaster has been designed to identify functions in the customer application layer, which are potentially affected by application of a patch to the Oracle system. The outcome of this research is that we will know more precisely what tests in any existing test suite could potentially be used to test the changes, and also whether there are no tests in the current suite that are applicable to testing changes caused by the current patch. In addition, the failure of a particular test case for certain inputs, and the absence of necessary test cases can also reinforce future re-engineering of the legacy application layer, i.e., the

forward development.

Impact analysis (or change impact analysis) is defined by Bohner and Arnold as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" [BA95]. Pfleeger and Atlee focus on the risks associated with changes. They state that impact analysis is: "the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and schedule". In our case, the consequence or the risk is the potential changed behaviour in the customer's application layer after patching; and we need to find the affected functions.

In this impact analysis project we set out to create a suite of automated tools which can be used to analyze the impact of patches on the customer application layer and derive a much reduced test suite based on the impacted parts of the system. First, it determines to the parts of the Oracle database and E-Business Suite that are physically changed by a patch. Then it finds the functions in the customer'a application layer that directly or transitively access the physically changed places. These are the potentially affected functions in the customer application. Based on these functions, it reduces the size of the test suite because it will only select tests relevant to the affected functions. As a side benefit, it may also suggest that necessary test cases are missing because there might be some affected functions with no test cases associated with them.

However, notice that for the process to be carried out successfully, we first need a knowledge base that stores the dependency relation among components of the system. Also, to begin impact analysis, we need the information regarding what actually was changed by a patch. For example, say function f_1 calls (depends on) function f_2. Now if a patch makes some change inside f_2 and we want to evaluate its impact, then we have to have two pieces of information at hand: the fact that f_2 is changed and the fact that f_1 is dependent on f_2. Finding the changes (modifications) and building up a dependency knowledge base are the two main foci of this thesis.

In computer science, program dependency graph generation is a very classic topic and substantial work has been carried out in this regard. However most of the research was confined to pure academic domains and even though many

of them resulted in some empirical success, none of them seems to have been applied to the kind of huge enterprise level system we are dealing with. Also, none of them dealt with a system in which a portion is legacy. In this thesis, we do not intend to propose any novel idea for finding a program dependency graph. Rather we do an empirical analysis on part of the real system (E-Business Suite) to build up a dependency graph (or relation) that would aid us in impact analysis; facing a number of practical challenges. In order to start the impact analysis, we also built a tool for detecting changes (modifications) between two successive versions of Java bytecode (*class*)[1] files. Our whole analysis process is static [Ern03] in nature, meaning that we will perform a worst case dependency analysis regardless of any particular execution trace. .

1.2 Thesis Organization

This thesis is organized as follows. In Chapter 2, we present the overview of the problem dealt with in this thesis. The foundation of the impact analysis project and a discussion of legacy systems are included. We also include our proposed solution approach and an overview of the tools we intend to build. Finally we describe, briefly, the specific subject matters of this thesis, namely modification finding and dependency graph generation.

In Chapter 3, we begin our discussion with some tools, concepts and techniques behind the generation of our dependency graph. This includes graphs and call graphs. We also discuss the notion of access dependency graph in an abstract manner. In addition, we discuss existing analysis techniques, like static and dynamic; and argue why we chose static analysis in our case. We finish the chapter by discussing some of the tools and storage techniques we used during the implementation of our tools.

In Chapter 4, we discuss much of the related work carried out in the fields of program dependency graph generation, impact analysis and modification detection. At the same time, we also argue why, despite being excellent ideas, we cannot apply most of those approaches in our empirical problem domain.

[1]Among the dozens of types of files in Oracle patches, we keep only *class* files in the scope of this thesis

Chapter 5 gives the process overview of our tools in short and also discusses some of the software engineering principles that we have followed during the design and implementation of our tools.

The next two chapters, Chapter 6 and Chapter 7, discusses the main subject matters of this thesis in detail. Chapter 6 discusses the process of finding modifications between two successive versions of Java *class* files. First we give a conceptual description of the possible kinds of modification and then we describe the implementation process we have followed in our tool.

Chapter 7 describes our dependency graph generation process in detail. We first discuss the concept of dependencies in the Java environment, keeping future impact analysis in mind. We also present a formal notion of access dependency analysis with an emphasis on Java. Then we go on and discuss the detailed implementation process of the dependency graph generation, along with many empirical challenges faced and some empirical software engineering decisions we had to make on our way. We furthermore discuss some performance statistics of the process as well. Finally we argue how many of the interesting impact analysis issues have been handled by our work presented in Chapter 6 and Chapter 7. Some parts of our implemented programs (with Java, PL/SQL) have been explained along with the discussion in Chapters 6 and 7. The complete code-base is proprietary to our project.

Chapter 8 is not directly a main subject matter of this thesis. Rather, it gives us some insight about maintaining the dependency graph over the course of time, as patches are applied. The information presented in this chapter is basically a reflection of what we have thought and planned regarding maintenance of the graph.

To conclude , in Chapter 9, we discuss the contribution and limitation of our work and suggest related future work that can be undertaken.

Chapter 2

Problem Definition

In this chapter we give an overview of the problem we deal with in this thesis. First we provide the background of the problem within the context of legacy systems, Oracle E-Business Suite and the hypothetical software testing approach of a typical customer whose system we are dealing with. Next we include a brief introduction of the impact analysis project (of which this thesis is a part) and the hierarchical structure of the project components. Finally, we present a brief description of the specific subject matter of this thesis.

2.1 Background

For the last 2 decades, computer technologies have been evolving quite rapidly. New technologies are being introduced very frequently. Very often a software system developed in one technology (or an older version of a certain technology) may find itself inefficient within a short span of time due to the introduction of new efficient technology (or a new more maintainable version of the same technology). Continuous technological advancement often lowers the business value of systems which had been developed over years through huge investments. For example a monolithic (single tier) system might have very little value compared to a three tier version of the same system. Also as [Cho05] mentions, the advancement in hardware technologies is much faster than that of software. So many software systems cannot take full advantage provided by newer hardware. Going back to the example just mentioned, a

monolithic system might not be able to take advantage of a three tier hardware system because the monolithic code might not have been written in such a way as to retain the possible presence of three tier technology in the future. Although more cost effective technologies are available, it is estimated that most systems in the IT industry either use legacy hardware platform or have one or more legacy software layers in the software system. Changing these systems into newer technologies would gain more efficient performance while keeping their functionalities intact. A study [Pig97] has indicated that most of these transformation cost a lot of money and hard work. We can define the systems which are running on older platforms or using an older software technology as legacy systems.

2.1.1 Legacy Systems

The Free On-Line Dictionary Of Computing (FOLDOC) defines a legacy system as, "A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and often despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify." [How98]. Wikipedia says, "A legacy system is an old method, technology, computer system, or application program that continues to be used, typically because it still functions for the users' needs, even though newer technology or more efficient methods of performing a task are now available. A legacy system may include procedures or terminology which are no longer relevant in the current context, and may hinder or confuse understanding of the methods or technologies used." [wik10a]. According to [Cho05], Bennett [Ben95] also gives some more more detailed characteristics of legacy systems:

- it may be written in assembly or an early version of a third generation language.
- probably developed using state-ofthe -art software engineering (programming pre 1968) techniques.
- many perform crucial work for the organization.

- generally large.
- generally hard to understand hence hard to maintain.

Many IT companies have legacy systems which (or part of which) were developed many years ago. From the definitions and characteristics of legacy systems, it is apparent that the software needed for those systems were developed in a time when modern sophisticated software engineering techniques were not available. These software were written taking efficiency as the main goal without considering concepts like modularity, low coupling, high cohesion that are an essential part of good large scale software development. So they are not easily portable into newer systems.

Besides, as pointed out in [Cho05], any legacy software systems are not properly documented. Over the years, coding style and documentation have changed in the area of software engineering. The style that was followed in legacy software might convey little meaning today. During the development of the legacy software, as the software was augmented with new features, improved for better performance, the documentation was not adjusted properly. To add insult to injury, in many of such systems we do not even have the source code of the software. We only have the compiled binaries. To make the situation even worse, the developers of the software might be unavailable (retired or moved somewhere else), so it becomes incredibly difficult to get proper understanding of the functionality of the code.

Software systems with a legacy layer may work fine for many years. But problems begin to occur when a change occurs inside the non-legacy layer(s). That change might be in the form of enhancing the non-legacy part by current developers, introduction of a new part (or a new layer) purchased from a third party vendor or applying patches to an already purchased portion to fix problems or cope with a changed requirement. These changes can cause serious wrong behaviour in the system, the lack of knowledge about the legacy layer being the prominent cause of this. As time evolves, this kind of problems becomes more and more prominent. Although very costly as mentioned earlier, eventually the system will need to be re-engineered the legacy layer with

newer technologies or with newer version and enhanced feature of the same technology.

But before the re-engineering process can be started, it needs to be known which functionalities in the legacy layer showed wrong or faulty behaviour after the changes to the non-legacy layer had been made. And to find those functionalities, companies having such systems usually have a test suite consisting of a huge number of test cases which are used to determine if anything goes wrong after a change in the non-legacy portion is made. But running that huge number of test cases needs a huge amount of investment because they might take weeks to complete. Also there is always a potential risk of missing a wrong behaviour, which is actually present in the system after the change has been made, because those tests are executed blindly. So a more efficient way of finding the potential wrong functionalities is necessary.

Before we move into further discussion about the problem, we present a brief overview of the Oracle E-Business Suite and a typical customer's system.

2.2 Oracle E-Business Suite

One of the significant parts of the customer's system that we are working with is the Oracle E-Business Suite. The E-Business Suite is a suite of Oracle's financial and various other applications to provide enterprise level solution for large scale customers. The following quote from [PA10] describes the E-Business suite in short:

> Oracle E-Business Suite is a software package that allows organizations to manage key business processes; it is known on the market by various names such as Oracle Enterprise Resource Planning (ERP), Oracle Apps, Oracle Applications, Oracle Financials, e-Biz and EBS (E-Business Suite). In this book we refer to it as either E-Business Suite, or Oracle Applications.

In the past, it was a common practice for businesses and organizations to develop in-house software to automate their business processes. Most of the software that was developed in-house largely matched the precise needs of the business. However, the fundamental business flows and processes such as accounting, procurement, human resource/employee management, and order management are based on common principles across all organizations. For example, most organizations require a system to make purchases from suppliers and a system to make payments to the suppliers, events known as transactions that need to be accounted for in the financial reporting. Enterprise Resource Planning (ERP) software prepackages different types of these functionalities into out-of-the-box software package, so that customers who purchase such software packages do not have to develop the same software applications time and again.

2.3 Customer's System

A typical customer has a large system consisting mainly of 3 parts:

- Customer's Legacy Application Layer
- Oracle E-Business Suite
- Oracle Database

Figure 2.1 shows the pictorial view of the customer's system.

At the very bottom, we have the Oracle database that consists of thousands of tables, views, stored procedures, functions, triggers etc. On top of that we have the Oracle E-Business Suite described in the previous section. Finally, on top of the E-Business Suite we have the customer's legacy application layer which might consist of forms, report generators, web pages etc.

From time to time, the customer receives patches from Oracle. Applying a patch can cause changes in the E-Business Suite and the Oracle database.

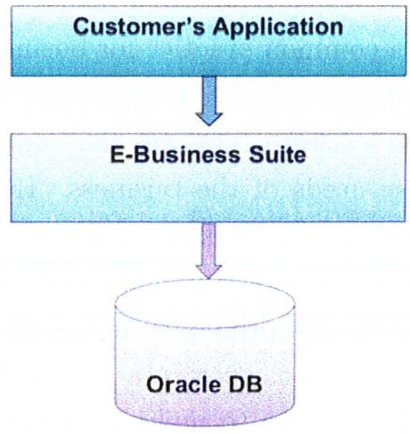


Figure 2.1: Customer's System Revisited

These changes might in turn cause the customer's application to behave differently which may be undesirable. So the parts of the customer's application which may give rise to such undesirable behaviours need to be detected. And detecting those parts is, in fact, the main goal of our project. Before proceeding further, we should discuss patches in general and Oracle patches in particular.

2.4 Patches

In computing, a patch is basically a piece of software to fix some problem or add some new functionalities to an existing software system. Wikipedia says, "A patch is a piece of software designed to fix problems[1] with, or update a computer program or its supporting data. This includes fixing security vulnerabilities and other bugs, and improving the usability or performance. Though meant to fix problems, poorly designed patches can sometimes introduce new problems." [wik10b]. FOLDOC says,"A temporary addition to a piece of code, usually as a quick-and-dirty remedy to an existing bug or misfeature. A patch may or may not work, and may or may not eventually be incorporated permanently into the program. Distinguished from a diff or mod by the fact that a patch is generated by more primitive means than the rest of the program; the classical examples are instructions modified by using

the front panel switches, and changes made directly to the binary executable of a program originally written in an HLL. Compare one-line fix." [How05]. One more definition from WiseGeek says, "As people begin to use a software program with frequency, they may note glitches or problems that were not observed during beta testing of the program. Alternately, older software can have compatibility issues with newer systems, or newer software may be incompatible with older systems. In these cases, and often to increase sales or use of software, programmers may create what is called a software patch, designed to fix small bugs, glitches, or address software-to-hardware or operating system compatibility issues." [EC10].

Since proprietary software authors withhold their source code, patches are usually distributed as binaries. Large patches may sometimes be called service packs or software updates [wik10b]. Software customers receive patches from the vendors and apply them to the system with the help of IT specialists in that particular fields and with the patch installation instructions as supplied by the vendors.

2.5 Oracle Patches

Oracle corporation periodically releases patches for their customers. In our case, we are only interested in patches related to the Oracle E-Business Suite. The detailed concept and process of patching procedures is described in [ora09]. The following subsections present a brief description of the patching concept and provide the gist of our findings from patch analysis, which has been done by my colleague Akbar Abdrakhmanov [1].

2.5.1 Reasons for patching

According to [ora09], throughout the course of an Oracle E-Business Suite life cycle, patches are applied for maintenance of the system. The reasons for this

[1] (in progress)

maintenance process include:

- Fixing an existing issue
- Adding a new feature or functionality
- Updating to a higher maintenance level
- Applying the latest product enhancements
- Providing interoperability to new technology stacks
- Determining the source of an issue
- Applying online help

2.5.2 Contents of a patch

Oracle patches are released in packaged bundle (zip) format. The details of the contents of a patch is described in [ora09]. What we are interested in is what sort of files a patch contains and how they can impact the system. My colleague Akbar Abdrakhmanov did some analysis on some sample patches after downloading them from metalink [Cor10]. His analysis revealed that files with the extensions mentioned in Table 2.5.1 can be present in a patch (of course not all them are present in one single patch).

.class	.dtd	.lct	.pdf	.pll	.sh	.xdf
.cmd	.fmb	.ldt	.pkb	.pls	.sql	.xgm
.ctl	.h	.mk	.pkh	.prop	.txt	.xml
.drv	.ildt	.o	.pl	.rdf	.wft	.xsl
.drvx	.jsp	.odf	.plb	.rtf	.wfx	

Table 2.5.1: Top 10 classes with highest number of transitive subclasses

Some of the file extensions mentioned here may be familiar to many, some other may be not. It is beyond the scope of this thesis to explain and analyze all of these file extensions. In this thesis, we will specifically deal with *.class* files, which will be the subject matter of the subsequent chapters.

2.6 Customer's Problem

As mentioned in section 2.3 our customer has a legacy application layer which is effectively a wrapper over the E-Business Suite and Oracle database. Whenever they apply a patch, it makes some changes or modifications to the E-Business Suite and/or to the database. And these changes can cause some of the functionalities of the legacy application layer to go wrong. To identify those wrongly behaving functionalities, they have a huge test suite which effectively covers all parts of the E-Business Suite and possibly also the database. Performing these tests have some drawbacks that were mentioned in section 2.1, but here is a quick recap:

- Executing all those tests are expensive both in term of money and time.
- Testing this way is essentially blind.
- Despite extensive testing, risk remains that an application will change or fail. Hence there is no guarantee of success.

So in order to save money, time and increase accuracy, they need a better and much more precise test suite, which may or may not be a proper subset of the much larger test suite that they are currently using. We are naming this project the impact analysis project.

The ultimate objective of the impact analysis project is to come up with the precise test suite. And this thesis is a one step advancement towards the goal of the project.

2.7 The Impact Analysis Project

In this section we give a brief overview of the impact analysis project and the tools to be developed in this project. The work presented in this thesis is a part of the tool suite architecture of the impact analysis project. A brief overview of our work is also presented in the next section.

2.7.1 Overview

As we stated earlier our customer has a legacy application layer which is a kind of wrapper over the E-Business Suite and Oracle database. And a modification in the E-Business Suite or in the database can cause some functionalities in the legacy layer to behave abnormally. Our ultimate goal is to provide the customer with suitable a test suite by which they can catch those abnormalities easily.

An important thing to note here is that unlike many other reverse engineering projects, we are not really interested in requirement extraction or producing a high level language equivalent of a system developed with a low level language. We are rather interested in finding the places of the legacy layer that can behave abnormally after the system has been patched. But for that we need to do a detailed impact analysis starting from a change (by a patch) in the non-legacy portion (E-Business Suite and Oracle database) and ending somewhere in the legacy portion of the system.

The impact analysis project basically does this detailed analysis in a static manner, with a view to ultimately finding the malfunctioning regions of the legacy layer of the customer's system and hence providing them with a much more concise and precise test suite than the one they are currently using.

The abstract steps of the impact analysis project are as follows:

- Identify those places in the database and the E-business Suite changed by a patch.
- Identify those places in the Customer's Application software that may access the changed places found by patch analysis.
- Select only tests relevant to those places.

Consider Figure 2.2 that shows the current testing approach of our customer. The test suite consists of lots of test cases and the coverage of the tests are shown by dotted lines. As can be understood, no specific criteria is applied to choose certain test cases or to exclude others.

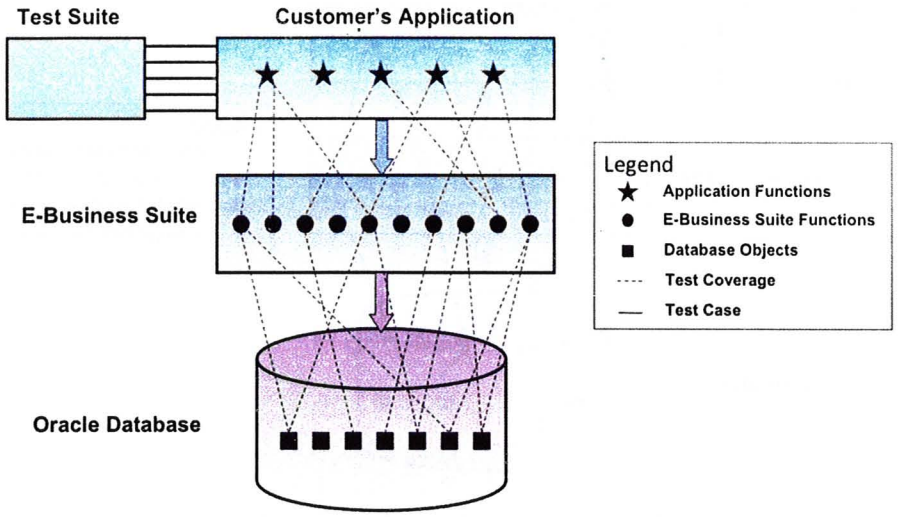


Figure 2.2: Current Testing Approach

Figure 2.3 shows the testing approach we plan to propose. The blackened squares and circles represent physically changed (by a patch) portions of the database and the E-Business Suite, respectively. And the grayed squares are circles represents the portions of the database and E-Business Suite that themselves didn't change, but are directly or transitively access those physically changed portions. So starting from the changed parts, if we can trace back to the customer's application layer (following the solid arrows in a reverse way) we can find those portions of the application layer that we need to worry about. These are indicated by the grayed stars in the figure. Note that the number of stars we need to be concerned about has reduced and so has the number of test cases. Also note that the second star from the left didn't previously have any test case associated with it. This indicates that we may also suggest new test cases along with reducing the size of the existing test suite.

Now for the tracing back task to be carried out successfully, we need to build up a dependency relationship, which is actually a caller-callee, or more precisely, an accessor-accessee relationship among the entities (methods, fields, database objects) of the system. In addition, we also need to be able to find out the physically changed parts of the system after patching so that the back

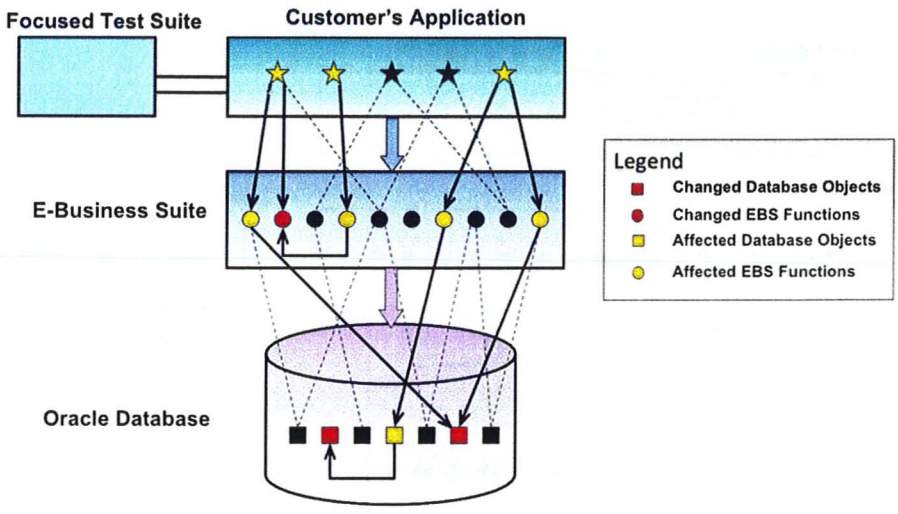


Figure 2.3: Proposed Testing Approach

tracing task can be started. These two issues are actually parts (among others) of our impact analysis project tool hierarchy discussed next.

2.7.2 Tool Hierarchy

The impact analysis project will generate a tool suite which will be used to find the precise test suite to apply after a patch. Figure 2.4 shows the tool hierarchy and interaction among the tools where arrows indicate a "used by" relationship.

It is beyond the scope of this thesis to describe what each component of the tool hierarchy does in detail. This thesis is mainly concerned with parts of the shaded regions in Figure 2.4 namely "Dependency Analyzer (Application)" and "Modification Finder (Application)". The following a short overview of each of the components of the tool hierarchy:

- **Patch Analyzer** is the tool for analyzing Oracle patches to find out which files would be modified or newly introduced by a patch. It internally consists of Oracle's *adpatch* command line tool and then our wrapper tool over it.

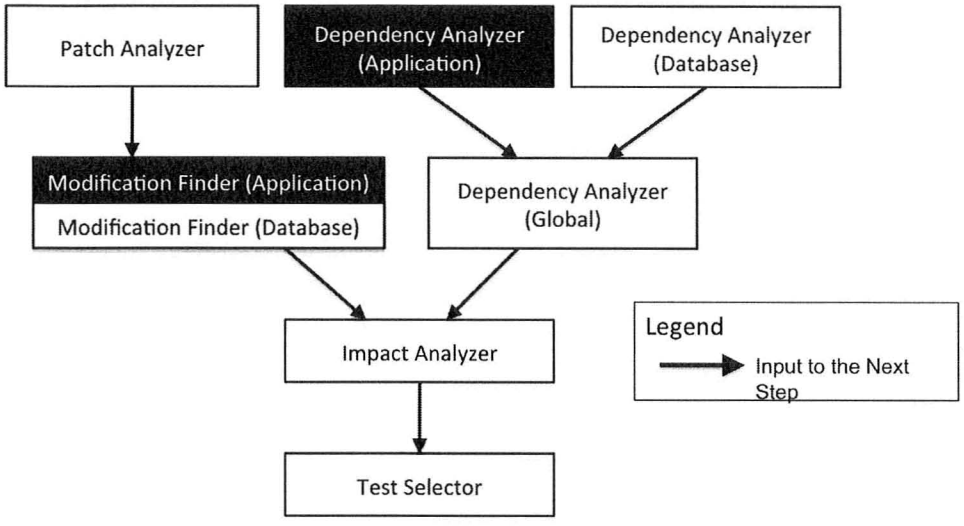


Figure 2.4: Tools Hierarchy

- **Modification Finder** is the tool that uses patch analyzer's output information to look for changes or modifications at an even finer grain (e.g., methods inside a Java *class* file), rather than at file level.
- **Dependency Analyzer (Application)** is the tool for analyzing dependencies among the entities of the application, and for building up a dependency relationship in a static manner. By application, we refer to both the E-Business Suite and the customer's legacy application layer.
- **Dependency Analyzer (Database)** is the tool for analyzing dependencies among the entities of the oracle database and building up a dependency relationship in a static manner.
- **Dependency Analyzer (Global)** will basically combine the dependency information from the above two dependency analyzers' output.
- **Impact Analyzer** is the tool for tracing back from the changes found from modification finder's output to the legacy application layer level using the dependency information from the global dependency analyzer.
- **Test Suite Selector** is the tool for finding a precise test suite based on

the information from the impact finder. It basically reduces the size of the huge test suite of the customer that they are currently using.

2.7.3 Dependency Analysis in the Application

Oracle's E-Business Suite is a huge application suite built using Java (and tools built on Java) as a front end and an Oracle database as the back end. The customer's legacy application layer, which is a wrapper over the E-Business Suite, is also supposedly developed in Java. In a system developed with an object oriented language like Java, there are entities like classes, methods, fields etc. And there exist complex dependencies among these entities. When a patch modifies any of these entities, we need to trace back in a bottom up fashion up to the legacy layer to find out what classes, methods or fields can be potentially affected by the changes. But for that we first need to build up a dependency relation among the entities of the system.

Since in the system, we only have the compiled binaries (in this case, Java bytecode files or class files) we need a way to analyze the bytecodes and build up that dependency relation. Since we don't know any execution sequence(s) conducted by the customer, dynamic analysis is not possible in our case. We have to stick to static worst case analysis. The details of all these has been described in subsequent chapters. Also, in many ways a static analysis is more suitable, since it provides a *safe* solution, and is not dependent on *coverage* questions inherent in a dynamic approach.

As stated earlier, an oracle patch can contain files with a whole variety of different extensions and it is beyond the scope of this thesis to analyze all those. In this thesis, we restrict ourselves to analyzing only class files.

2.7.4 Finding Modifications

Aside from finding the dependencies, another relatively small focus of this thesis is in finding the modifications made by a patch. The patch analysis tells us which files have been modified. But for finding the modifications at even a finer level than a file (e.g. a method inside a class file) we need to perform additional work which is described in Chapter 6.

Just like the case of dependency analysis, it is beyond the scope of this thesis to find modifications in all kinds of files. We restrict ourselves only to class files.

2.8 Summary

In this chapter, we discussed our problem background, our proposed solution approach within the complete impact analysis project, and a brief overview of the specific subject matters of this thesis.

The next chapter presents the concepts, techniques and tools that we use in our work.

Chapter 3

Tools and Techniques

There has been increasing interest in the application of sophisticated program analysis techniques to software development and maintenance tools. Such tools include those which are used for program understanding, verification, testing, debugging reverse engineering etc. In this chapter we present and describe some analysis tools and techniques which are relevant to our impact analysis project.

3.1 Graphs

Graphs have been used extensively to model many problems that rise in the fields of computer science and software engineering. Specially in software engineering, Call Graph, Control Flow Graph, Data Flow Graph, Component Graph etc. give a better analytical approach to understand and characterize software architecture, static and dynamic structure and meaning of the programs [Cho05]. A diagrammatic view (by graphs) of the structure of the code is always an excellent way present the issues related to software engineering analysis. That is why graphs are always preferred by software engineers and researchers to understand, re-engineer and analyze codes.

A number of graph analysis techniques are available for software engineering applications. Control Flow Analysis, Data Flow Analysis, Call Graph Analysis, Analysis using Component Graph are some of them. In Control Flow Analysis, a Control Flow Graph is used to analyze and understand how

the control of the program is transferred from one point to another. Similarly Data Flow Analysis uses Data Flow Graphs to show and analyze data dependencies among the instructions of the program. Component graphs identifies the components of a program, shows the use relations among those components are very useful in software architecture identification and recovery. Call Graph Analysis uses call graphs to detect calling dependency relations among entities of the environment.

For reasons discussed in subsection subsection 3.1.3, we use a new notion which we name Access Dependency Analysis, which is based on an Access Dependency Graph (an Extension of a Call Graph), for analyzing the dependency relationship among entities in the Java environment. In the following subsections, we describe Call Graph and Access Dependency Graph in detail.

3.1.1 Call Graph

A call graph is a directed graph $G = (N, E)$ with a set of nodes N and a set of edges $E \subseteq N \times N$. A node $u \in N$ represents a program procedure and an edge $(u, v) \in E$ indicates that procedure u calls procedure v. Consider the call graph in Figure 3.1. It has a set of nodes $N = \{a, b, c, d, e\}$ and e set of edges $\{(a, b), (a, c), (b, d), (c, d), (c, e)\}$.

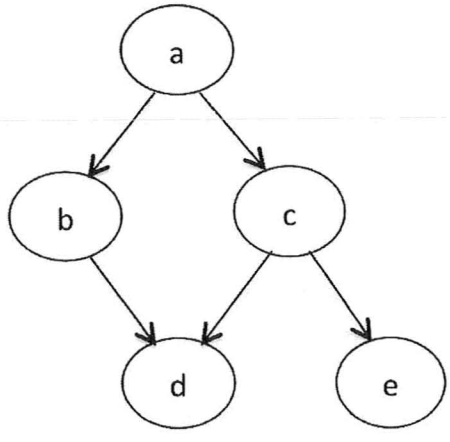


Figure 3.1: Call Graph

3.1.2 Access Dependency Graph

The simple notion of call graph described in the previous subsection works well in traditional non-object oriented languages like C. But in an object oriented language like Java, where methods (procedures) are encapsulated inside classes and those classes can have fields in addition to methods, the notion of a call graph is far more complicated. Also features like inheritance, dynamic binding etc, can introduce implicit dependencies on methods or fields which are not explicitly present in the source code or even in the compiled binary bytecode. For example, consider class B which extends class A and overrides A's method $m()$. Now there is an explicit call from class C's method $c()$ to class A's method $m()$, and due to dynamic binding this call might actually result in a call to class B's method instead of class A's method $m()$.

Due to empirical issues that will be discussed in detail in Chapter 7, we introduce a new notion called *Access Dependency Graph* and define it as follows: An access dependency graph is a directed graph $G = (N_m, N_f, E)$ with a set of method nodes N_m, a set of field nodes N_f and a set of edges $E \subseteq (N_m \cup N_f) \times (N_m \cup N_f)$. A node $m \in N_m$ indicates a method node and is of the form *ClassName* : *MethodName*. A node $f \in N_f$ indicates a method node and is of the form *ClassName* : *MethodName*. An edge (m, e) in E (where $m \in N_m$) may indicate one of two kinds of dependency:

- An explicit method call from method m to method e if $e \in N_m$, or an explicit access of a field e from method m if $e \in N_f$.
- An implicit dependency from method m to method e where an explicit call from somewhere to method m may actually result in a call to method e due to dynamic binding.

In Chapter 7, we will discuss the notion of call graph in Java context, the empirical reasons why we switched from call graph analysis to access dependency analysis and the details of access dependency analysis itself. We will also show that for static fields there is a further kind of edges we include in the graph.

Consider the access dependency graph in Figure 3.2. It has a set of method nodes $N_m = \{D : d(), C : c(), E : e(), A : m(), B : m()\}$, a set of field nodes

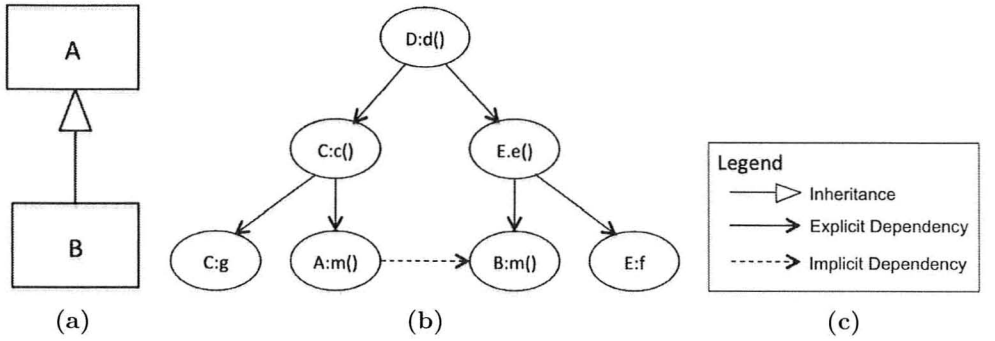


Figure 3.2: (a) Inheritance of class B from class A. (b) Access Dependency Graph. (c) Meaning of the arrows.

$N_f = \{C : g, E : f\}$ and a set of edges $\{(D : d(), C : c()), (D : d(), E : e()), (C : c(), A : m()), (C : c(), C : g()), (E : e(), B : m()), (E : e(), E : f), (A : m(), B : m())\}$. The first six edges are due to explicit dependency and the last one is due to implicit dependency.

The reader might wonder why in Figure 3.2 there is an edge $A : m() \rightarrow B : m()$ instead of $C : c() \rightarrow B : m()$. The reason for this will be discussed in detail in Chapter 7.

3.1.3 Reasons for Choosing Access Dependency Analysis

Although conducting a detailed control flow and data flow analysis seems more appropriate for our problem, we stick to access dependency analysis for the following reasons:

- Oracle's E-Business Suite has a huge file system consisting of almost 170,000 class files and thousands of jar files (which contain thousands of class files in themselves), along with other files. Conducting a detailed control flow or data flow analysis is currently not feasible in such a large domain. Moreover, since we do not have access to a customer's test suite or any particular execution sequence, conducting the control or data flow analysis dynamically is not feasible either. As will be discussed in

Chapter 4, the related works in this area that uses control or data flow analysis focuses on much smaller domains than ours.

- Since the test suite we are concerned with consists of tests at the integration level and not at the unit testing level, the test suite is presumably very sparse. That is, it is very unlikely that multiple tests will cover one single procedure (Java method) in the E-Business Suite. So a control flow or data flow analysis does not seem to bring us much better results than an access dependency analysis.

- In future, however, when we have a much better knowledge about our problem domain and have access to a customer's test suite, we might be able to conduct control flow or data flow analysis on parts of the system anyway.

3.2 Static versus Dynamic Analysis

Static and dynamic analyses arose from different communities and evolved along parallel but separate tracks [Ern03]. Traditionally, they have been viewed as separate domains, with practitioners or researchers specializing in one or the other. Furthermore, each has been considered ill-suited for the tasks at which the other excels.

Static analysis examines program code and reasons over all possible behaviours that might arise at run time [Ern03]. Compiler optimizations are standard static analyses. Typically, static analysis is conservative and sound. Soundness guarantees that analysis results are an accurate description of the program's behavior, no matter on what inputs or in what environment the program is run. Conservatism means reporting weaker properties than may actually be true; the weak properties are guaranteed to be true, preserving soundness, but may not be strong enough to be useful. For example, in our access dependency graph, the conservative analysis might report dependencies (calls in this case) from methods a, b, c to method d whereas for a particular user, only the dependency from method a to method d might be relevant, the other two are not.

Static analysis operates by building a model of the state of the program, then determining how the program reacts to this state [Ern03]. Because there are many possible executions, the analysis must keep track of multiple different possible states. It is usually not reasonable to consider every possible run-time state of the program; for example, there may be arbitrarily many different user inputs or states of the runtime heap. Therefore, static analyses usually use an abstracted model of program state that loses some information, but which is more compact and easier to manipulate than a higher-fidelity model would be. In order to maintain soundness, the analysis must produce a result that would be true no matter the value of the abstracted-away state components. As a result, the analysis output may be less precise (more approximate, more conservative) than the best results that are in the grammar of the analysis.

Dynamic analysis operates by executing a program and observing the executions [Ern03]. Testing and profiling are standard dynamic analyses. Dynamic analysis is precise because no approximation or abstraction need be done: the analysis can examine the actual, exact run-time behavior of the program. There is little or no uncertainty in what control flow paths were taken, what values were computed, how much memory was consumed, how long the program took to execute, or other quantities of interest. Dynamic analysis can be as fast as program execution. Some static analyses run quite fast, but in general, obtaining accurate results entails a great deal of computation and long waits, especially when analyzing large programs.

The disadvantage of dynamic analysis is that its results may not generalize to future executions [Ern03]. There is no guarantee that the test suite over which the program was run (that is, the set of inputs for which execution of the program was observed) is characteristic of all possible program executions. Whereas the chief challenge of building a static analysis is choosing a good abstraction function, the chief challenge of performing a good dynamic analysis is selecting a representative set of test cases (inputs to the program being analyzed). (Efficiency concerns affect both types of analysis.) A well-selected test suite can reveal properties of the program or of its execution context; failing that, a dynamic analysis indicates properties of the test suite itself, but it can be difficult to know whether a particular property is a test suite artifact

or a true program property.

3.3 Why We Chose Static Analysis

Despite the fact that dynamic analysis is gaining increasing popularity as many research papers like [Ern03] pointed out, for the impact analysis project we restrict ourselves to conservative static analysis for the following reasons:

- Dynamic analysis is not possible unless the full domain, on which programs are run to gather execution sequences, is available. As mentioned in Chapter 2, currently we only have access to the compiled Java bytecode files of the E-Business suite, neither to the customer's test suite nor to any real execution sequence. So for the moment and possibly for a good amount of time in the future, dynamic analysis is beyond our scope.

- As mentioned in the previous section, one of the disadvantages of dynamic analysis is its inability to generalize to future executions. Since our customer's system is periodically modified by patches, it is more and more likely to change its behaviour over the course of time. So the test cases need to be run again and again in order to make sure that the system is behaving properly. Static analysis, on the other hand, being sound, as we show later in Chapter 8, is easy to conduct over changes made by patches.

- Most importantly, since our ultimate goal is to reduce the customer's test suite size and to propose additional tests if necessary, dynamic analysis would practically be equivalent to executing all their tests, which defeats our whole long term purpose.

In future, however, we may apply dynamic analysis as we gain access to the customer's test suite. Also we may combine static and dynamic analysis for getting better results. As [Ern03] pointed out, Static or dynamic analyses can enhance one another by providing information that would otherwise be unavailable. Performing first one analysis, then the other (and perhaps iterating) is more powerful than performing either one in isolation.

3.4 Implementation Tools

The tool suite architecture of the impact analysis project is hierarchical (meaning that there are sub-tools with one's output serving as another's input), and will become more complex as it grows. That is why it is important to structure it well. In our project we have to deal with Java bytecode a lot because the E-Business Suite and the patches contain Java bytecode files (class files). To analyze them, we use the Java language itself for a number of reasons:

- Java is a well structured and extensively used Object Oriented Programming language. It is easier to cope with many software engineering related issues with Java rather than lower level languages like C or C++.
- There is good open source project support and many forums related to Java available on the web.
- There are a number of existing bytecode analysis tools written in Java itself.

As part of our work, we also use Oracle database to handle some critical issues which will be discussed in Chapter 8. In the next two subsections we briefly describe two of the bytecode analysis tools that we have used (only the second one was used in the final version).

3.4.1 Dependency Finder

Dependency Finder [Tes10a, Tes10b] is a suite of tools for analyzing compiled Java code. At the core is a powerful dependency analysis application that extracts dependency graphs and mines them for useful information. This application comes in many forms promoting ease of use, including command-line tools, a Swing-based application, a web application ready to be deployed in an application server, and a set of Ant [Fou10] tasks.

Among the suite of tools, the specific tools that we used for our purpose are ClassReader and DependencyExtractor. Both of these tools generate XML files that can be used for next order analysis. ClassReader's XML is a one to

one XML representation of a certain class file, whereas DependencyExtractor's XML specifically encompasses the dependency information. However, although DependencyExtractor is an excellent tool for extracting depencdency relationships from class (and jar, zip as well) files, again, due to some empirical issues (which are discussed in Chapter 7), we decided to use ClassReader's XML to build the repository for our analysis.

3.5 Advantage of Using XML and Database

Having an XML representation of a certain java class file has a number of advantages over having any other kind of representation (e.g. Java source file). XML can always be used as a well formatted input for next order processing. XML is also a kind of data repository for useful information. And in almost all widely used programming languages like Java, there is a rich built in facility to parse and play with XML.

We use the Oracle database as a repository for the extracted required information from the XML. Technically, we could have used any other database system instead of Oracle; but we have good access to our university's Oracle server, and decided to use it. Storing the information in the database also has a number of advantages. For example, we can apply intelligent queries on the database tables to extract certain information. In the future, the data might also be mined for extracting hidden patterns in the dependency relationship.

3.6 Summary

In this chapter, we presented the theoretical concepts, analysis techniques and implementation tools used in our impact analysis project. Their empirical application will be discussed in detail in Chapters 6 and 7.

The next chapter presents much of the related work in the fields of program dependency graph generation, impact analysis and test selection; along with their their applicability issues in our problem domain.

Chapter 4

Related Work

There has been extensive work in the area of program dependence graph construction, interprocedural analysis and impact analysis over the last two decades. In this chapter we discuss the main focus of some of the related works and their applicability issues in our specific problem domain. For the sake of clarity we divide the related work into three categories:

- Works related to program dependence graphs and interprocedural analysis
- Works related to impact analysis and test case selection
- Works related to finding differences between subsequent versions of programs

In the next three sections we discuss each of the above.

4.1 Work Related to Program Dependence Graphs and Interprocedural Analysis

The notion of dependence graphs (call graphs, control flow graphs, data flow graphs) was introduced in Chapter 3. As a recap, a call graph is a directed graph that represents the calling relationship among procedures in a computer

program. Interprocedural analyses enable optimizing compilers to more precisely model the effects of procedure calls, potentially resulting in substantial increases in application performance. Applying interprocedural analysis to programs written in object-oriented or functional languages is complicated by the difficulty of constructing an accurate program call graph. Work related to these areas dates back to the 70's.

Ryder [Ryd79] introduced the notion of call graph as an acyclic graph to reduce the dynamic relation among procedures to a static data representation. Its main application domain was in FORTRAN. Callahan's extension to Ryder's work [CCHK90] was to support recursion. However, these works were not carried out in object oriented domains and so they did not face many challenges that arise in an object oriented context.

Grove *et al.* [GDDC97,GC01] presented a parameterized algorithmic framework for call graph construction in the presence of dynamic binding. They used this framework to describe and to implement a number of well-known and new algorithms. They then empirically assessed these algorithms by applying them to a suite of medium-sized programs written in Cecil [cec] and Java, reporting on the relative cost of the analyses, the relative precision of the constructed call graphs, and the impact of this precision on the effectiveness of a number of interprocedural optimizations. Their work, however, was basically concerned with comparing the precision of existing interprocedural analysis algorithms. Many of those algorithms [CC77, AM95, Ste] dealt with complex issues like recursion and mutually recursive definitions and the possible infinitely nested calling sequences in them etc, which is not really relevant to our case. Also, these works were rather generic approaches for call graph generation and inter procedural analyses and were not carried out keeping future impact analysis in mind. In addition, the application in the domains in which they assessed their work were several orders of magnitude smaller than ours.

Ferrante *et al.* [FOW87] represented an intermediate program representation, called the program dependence graph (PDG), that makes explicit both the data and control dependences for each operation in a program. Harrold *et al.* [HMR93,Sin01] presented techniques for constructing program dependence graphs using control flow and data dependence information. However as we

mentioned in Chapter 3, due to empirical issues, a detailed control or data flow analysis is not currently feasible in our problem domain.

Harrold *et al.* [HLL+95] developed a system called Aristotle that supports program analysis information and supports the development of software engineering tools. They implemented parsers for C and C++ that gather control flow, local data flow and symbol table information. This system, though promising, is not suitable for our case because it doesn't have support for Java and we are not doing a detailed control or data flow analysis.

One point worth mentioning here is that the dependency graph we intend to generate has to be such that impact analysis can be carried out successfully. We don't really care about how generic or how complex the graph is. For example, there might be mutually recursive definitions that might produce infinite call sequences between them but all we need to know in our case is that they call each other, so that if any of them is changed we can track the change backwards up to the callers. So, many of the above ideas, despite being excellent in general or for other specific problem domains, is not quite suitable in our case; because they were carried out for other purposes rather than keeping possible future impact analysis in mind. Also notice that none of them deals with a domain where a portion of the system is legacy.

The Soot framework [LBL+10], is a set of Java Application Programming Interfaces (API) for manipulating Java code in various forms. It analyzes complete applications, by first reading all class files that are required by an application, starting with the main root class and resursively loading all classes used in each loaded class. As each class is read it is converted into the Jimple intermediate representation. After conversion each class is stored in an instance of a `SootClass`, which in turn contains information such as its name, its superclass, a list of interfaces that it implements, and a collection of `SootFields` and `SootMethods`. Vijay *et al.* [SHR+00] show how they used Soot to conduct different kinds of analysis like Class Hierarchy Analysis (CHA), Rapid Type Analysis (RTA), Variable Type Analysis (VTA) etc. to resolve the virtual method calls and to construct a call graph with minimum possible number of edges. We will discuss more about the virtual method call problem (what we

also call inheritance problem) in Chapter 7. We will also discuss this technique and show with some empirical studies, why all these techniques are not suitable for our specific problem domain.

JAnalyzer [Bod03], developed by Eric Bodden aids program development by construction of call graphs by state of the arts analyses, visual representation of interdependencies among methods and comprehensible view of even very large call graphs. It uses Class Hierarchy Analysis (CHA) and Variable Type Analysis (VTA). The set of static types of the possible call targets are retrieved in order to perform the appropriate query on the call graph using internal SOOT mechanisms. However, up to now, JAnalyzer could only give us static call graphs in a graphically visual form, i.e., its output is not suitable for higher order analysis.

Profile Viewer [Whi09] reads profiling information produced by the Java interpreter and various flavours of the gprof tool and displays it for easy interpretation. But the problem of Profile Viewer is it can only give us dynamic call graphs. Since we are doing static analysis this is not suitable for our case.

Dependency Finder [Tes10a] is a suite of tools for analyzing compiled Java code. At the core is a powerful dependency analysis application that extracts dependency graphs and mines them for useful information. This application comes in many forms, including command-line tools, a Swing-based application, a web application ready to be deployed in an application server, and a set of Ant [Fou10] tasks. This tool (actually two subtools inside the tool suite) can give us two kinds of XML files - one is a one-to-one XML representation of a class (bytecode) file and the other is specific to dependency information only. We use the former for our analysis and that is the heart of Chapter 7.

4.2 Work Related to Impact Analysis and Test Case Selection

Although impact analysis and test case selection is not the main subject matter of this thesis, they are heavily dependent on how we construct the access dependency graph. While constructing the dependency graph, we have to keep in mind how the impact analysis and test case selection will be carried

out later on. Below we describe some of the works related to impact analysis and test case selection.

Over the last two decades, there have been a good number of works and articles published on software change impact analysis (IA). Bohner and Arnold [BA95] identify two classes of IA, traceability and dependency IA. In traceability IA, links between requirements, specifications, design elements, and tests are captured, and these relationships can be analysed to determine the scope of an initiating change. In dependency IA, linkages between parts, variables, logic, modules etc. are assessed to determine the consequences of an initiating change. Dependency IA occurs at a more detailed level than traceability IA. From their definition it is clear that it is dependency IA that applies to our problem domain where our logical modules are methods and fields inside Java classes.

Just like program dependence graph generation techniques, dependency impact analysis can also be either static or dynamic. In our case only static impact analysis is possible. We discuss some of the works related to both of these techniques, below.

Static impact analysis techniques (e.g. [BA95,LMS97,PA06,RST+03,TM94]) identify the impact set – the subset of elements in the program that may be affected by the changes made to the program. Apiwattanapong *et al.* [Api05] pointed out that static impact analysis algorithms often come up with too large impact sets due to their over conservative assumption and might turn out to be effectively useless. For example, regression testing techniques that use impact analysis to identify which parts of the program to retest after a change would have to retest most of the program. [Api05] also points out a two fold problem with sound static impact analysis. First, they consider all possible behaviors of the software, whereas, in practice, only a subset of such behaviours may be exercised by the users. Second, and more importantly, they also consider some impossible behaviours, due to the imprecision of the analysis. Therefore, recently, researchers have investigated and defined impact analysis techniques that rely on dynamic, rather than static, information about program behaviour [LR03a, LR03b, OAH03, BDSP04]. The dynamic information consists of execution data for a specific set of program executions,

such as executions in the field, executions based on an operational profile, or executions of test suites.

[Api05] defines dynamic impact set to be the subset of program entities that are affected by the changes during at least one o the considered program executions. `CoverageImpact` [OAH03] and `PathImpact` [LR03a, LR03b] are two well known dynamic impact analysis techniques that uses dynamic impact sets. `PathImpact` works at the method level and uses compress execution traces to compute impact sets. `CoverageImpact` also works at the method level but it uses coverage, rather than trace, information to compute impact sets. The coverage information for each execution is stored in a bit vector that contains one bit per method in the program. If a method is executed in the execution considered, the corresponding bit is set; otherwise it remains unset. In [OAL+04], the precision and performance of `CoverageImpact` and `PathImpact` have been compared. `PathImpact` turns out to be more precise but more costly in terms of time and space.

[Api05] made some fundamental observations about the essential information that is required to perform dynamic impact analysis. Using those observations, they introduced a notion of *Execute After* sequences (EA) which is based on the principle that to identify the impact set for a changed entity e we must include all program entities that are executed after e in the considered program execution.

Now considering our problem domain, as we mentioned earlier, since we don't have access to our customer's test suite or any execution sequence at this moment, dynamic analysis and hence dynamic impact analysis is not possible in our case. Also another subtle difference of our work with the work presented above, is that we are not really interested in finding the total impact set. We are only interested in finding the functions in the customer's application layer that may lead to a changed behaviour after a change has been made by the patch somewhere in the Oracle's E-Business Suite. Considering long term issues, doing a static impact analysis has certain advantages; for example, tracing back from the changes, some function might be found in the customer's application layer that does not show changed behaviour for certain executions but may be in the future show changed behaviour. Keeping in mind that we

want to minimize the customer's risk of patching, this kind of conservative analysis should bring us good results in the long run. Moreover, in the future, when we might have access to a customer's test suite and execution sequences, we will be able to conduct dynamic impact analysis as well and the outcome from the static analysis can be a good source of information at that time.

Chianti [RST+03, RST+04] is a tool for change impact analysis for Java that is implemented in the context of the *Eclipse* [ecl] environment. *Chianti* analyzes two versions of an application and decomposes their difference into a set of atomic changes. Change impact is then reported in terms of affected (regression or unit) tests whose execution behaviour may have been modified by the applied changes. For each affected test, Chianti also determines a set of affecting changes that were responsible for the test's modified behaviour. This latter step of isolating the changes that induce the failure of one specific test from those changes that only affect other tests can be used as a debugging technique in situations where a test fails unexpectedly after a long editing session. Their analysis comprises the following steps [RST+04]:

1. A source code edit is analyzed to obtain a set of interdependent atomic changes $\mathcal{A}$, whose granularity is (roughly) at the method level. These atomic changes include all possible effects of the edit on dynamic dispatch.

2. Then, a call graph is constructed for each test in $\mathcal{T}$. In [RST+03], they use static call graphs and in [RST+04] they use dynamic call graphs.

3. For a given set T of (unit or regression) tests, the analysis determines a subset $\mathcal{T}'$ of $\mathcal{T}$ that is potentially affected by the changes in A, by correlating the changes in A against the call graphs for the tests in T in the original version of the program.

4. Finally, for a given test $t_i \in \mathcal{T}$, the analysis can determine a subset $\mathcal{A}'$ of $\mathcal{A}$ that contains all the changes that may have affected the behaviour of t_i. This is accomplished by constructing a call graph for t_i in the edited version of the program, and correlating that call graph with the changes in $\mathcal{A}$.

Once again, since we do not have to the customer's test suite, we cannot follow the *Chianti* approach. Also since we also want to suggest additional test cases in addition to finding a subset of existing tests that needs to be rerun, only using call graphs for the tests is not sufficient for our case either. Another major issue is that the size of the domain over which they tested their techniques is less than 1000 classes, whereas ours has almost 230,000 classes. *Chianti*, however, encompasses information about types of atomic changes that were extremely useful to us. They pointed out 16 kinds of atomic changes that we had to think about. In Chapter 7, we will show how we build our access dependency graph keeping in mind all these kinds of changes.

As far as test selection is concerned, a fair amount of work have been done in that area too. By test selection, here we mostly mean regression test selection. Regression testing is the process of validating modified software to provide confidence that the changed parts of the software behave as intended and that the unchanged parts of the software have not been adversely affected by the modifications. Because regression testing is expensive, researchers have proposed techniques to reduce its cost. One approach reduces the cost of regression testing by reusing the test suite that was used to test the original version of the software. Rerunning all test cases in the test suite, how- ever, may still require excessive time. An improvement is to reuse the existing test suite, but to apply a regression- test-selection technique to select an appropriate subset of the test suite to be run. If the subset is small enough, significant savings in time are achieved. To date, a number of regression test selection techniques have been developed for use in testing procedural languages (e.g., [Bal98, CRa94, LW91, RH97, FF97, LW92]) and for use in testing object-oriented languages (e.g., [HLK+97, KGH+94a, KGH+94b, RHD00, AK97]). A *safe* regression test selection technique is one that, under certain assumptions, selects every test case from the original test suite that can expose faults in the modified program [RH96]. Several safe regression test selection techniques (e.g., [Bal98, CRa94, RH97, FF97, RHD00]) exist. These techniques

use some representation of the original and modified versions of the software to select a subset of the test suite to use in regression testing. Empirical evaluation of these techniques indicates that the algorithms can be very effective in reducing the size of the test suite while still maintaining safety [BRR01, GHK+01, KPR00, RH98, RHD00, FF97].

In case of object oriented languages, a number of regression test selection techniques have been developed [RHD00, AK97, HJL+01]. Rothermel, Harrold and Dedhia's algorithm [RHD00] was developed for only a subset of C++, and has not been applied to software written in Java. White and Abdullah's approach [AK97] also does not handle certain object-oriented features, such as exception handling. Their approach assumes that information about the classes that have undergone specification or code changes is available. Using this information, and the relationships between the changed classes and other classes, their approach identifies all other classes that may be affected by the changes, and it is these classes that need to be retested. White and Abdullah's approach selects test cases at the class level and, therefore, can select more test cases than necessary.

Harrold *et al.* [HJL+01] presents the first safe regression test selection technique for Java that efficiently handles the features of Java language. Our technique is an adaptation of Rothermel and Harrold's graph-traversal algorithm [RH97, RHD00], which uses a control-flow-based representation of the original and modified versions of the software to select the test cases to be rerun. They use the notion of *coverage matrix* and *dangerous entity*. Assuming P and P' to be the actual and modified version of a program, respectively, the coverage matrix records which entities of P are executed by each test case in a test suite T. A dangerous entity is a program entity e such that for each input i causing P to cover e, $P(i)$ and $P'(i)$ may behave differently due to differences between P and P'. Rothermel and Harrold describe a regression test selection technique that uses a control flow graph (CFG) to represent each procedure in P and P' and uses edges in the CFGs as potential dangerous entities [RH97]. Dangerous entities are selected by traversing in parallel the CFGs for P and the CFGs for P'; whenever the targets of like-labeled CFG edges in P and P' differ, the edge is added to the set of dangerous entities. After dangerous edges

have been identified, the system uses the dangerous entities and the coverage matrix to select the test cases in T to add to T'.

Although [HJL+01] seems to be pretty strong technically, their approach is not suitable for us because we are using static analysis and do not have access to the test suite now. Also, we are not executing a customer's program ourselves, and the sizes of the applications in the domain in which they carried out their empirical studies are pretty small compared to ours. They used *JEdit* which has less than 4,000 methods whereas ours has almost 3 million methods. Moreover, their main focus was the control flow inside procedures which pertains more to unit tests rather than unit tests. As mentioned in Chapter 2, a customer's test suite is presumably an integration test suite and is pretty sparse. Due to the huge size of our problem domain, we are not doing a detailed control flow inside each procedure (method), rather we are building an access dependency relationship whose granurality is at the method level. However, we can use the notion of dangerous edges from [HJL+01] in a slightly different manner in our case as will be described in Chapter 6.

Orso *et al.* [OSH04] presented a new regression test selection algorithm for Java programs that handles the object-oriented features of the language, and is safe and precise. The algorithm consists of two phases: partitioning and selection. The partitioning phase builds a high-level graph representation of programs P and P' and performs a quick analysis of the graphs. The goal of the analysis is to identify, based on information on changed classes and interfaces, the parts of P and P' to be further analyzed. The selection phase of the algorithm builds a more detailed graph representation of the identified parts of P and P', analyzes the graphs to identify differences between the programs, and selects for rerun test cases in T that traverse the changes. Their base idea is effectively the same as in [HJL+01] but due to the two phases, they claim and show with some empirical studies that this technique scales up to large software systems. However, the largest domain they applied their technique to has over 2,400 classes which is still way beneath the number of classes in our domain – almost 230,000. And also, due to the same reasons mentioned in the previous paragraph, [OSH04] is not quite suitable for our problem domain now.

[OSH04], however has the idea of partitioning the program in components according to inheritance hierarchy and applying edge-level test selection. In Chapter 7, we are going to show why this idea is not quite suitable for tackling the inheritance problem (virtual method call problem) in our problem domain.

4.3 Work related to finding differences between subsequent versions of programs

Finding differences between two successive versions of a program (although in our case, just class files) is one of the subject matters of this thesis because the difference information is one of the two inputs to the impact analysis, the other being the dependency graph. A number of works have been conducted in this area as well.

Impact analysis identifies the parts of a program that are affected by changes and, thus, requires knowledge of the location of such changes. Many regression test selection techniques (e.g., [OSH04, RH97]) use change information to select test cases to be rerun on modified versions of the software.

There are a number of existing techniques and tools for computing textual differences between files (e.g., the UNIX diff utility [Mye86]). However, these techniques are limited in their ability to detect differences in programs because they provide purely syntactic differences and do not consider changes in program behaviour indirectly caused by syntactic modifications.

Other existing differencing techniques are specifically targeted at comparing two versions of a program (e.g., [Hor90, JL94, MPW00]), but they are not suitable for object oriented code.

Apiwattanapong *et al.* [AOH04] presented a technique for comparing object oriented programs that identifies both differences and correspondences between two versions of a program. The technique is based on a representation that handles object oriented features and, thus, can capture the behaviour of object oriented programs.

Despite containing good ideas, the above techniques are not suitable in our case. As mentioned in Chapter 1, an oracle patch comes in a zip bundle format containing new versions of modified class files and new class files (along with

other files). And because of the huge size of the E-Business Suite, keeping two versions of the entire system is not feasible for us (one with the older version of files and the other with the newer version). Also, we are not really interested in finding the modified behaviours because we are not executing the customer's program ourselves. The thing that we are interested in is to find the functions in the customer's application layer that may initiate a changed behaviour, i.e., that leads a call path to a changed function (method). So we actually need to find what methods have been modified, added or deleted between two successive versions of a class file.

JDiff [Doa07] is a Javadoc doclet which generates an HTML report of all the packages, classes, constructors, methods, and fields which have been removed, added or changed in any way, including their documentation, when two APIs are compared. This is very useful for describing exactly what has changed between two releases of a product. Only the API (Application Programming Interface) of each version is compared. Also it needs the whole API information as input; so this is not suitable for us.

Jar Compare Tool [dif] is another good tool that allows comparing of classes inside Java JAR archives and displaying specific differences in class files. The tool performs deep comparison - it decompiles the class files and displays differences in specific code lines. However, since it has to work with jar files and we are dealing with class files, we cannot use this tool for our purpose as well.

ClassClassDiff is a tool inside the Dependency Finder toolset [Tes10a] that almost does what we need, but not quite. Its limitation is that it can only show API differences between 2 classes, but cannot report any change in case of any modified classes.

Considering all these, we decided to write our own tool for determining difference between two class files according to our need. We basically use the XML representation of class files, generated by the *ClassReader* tool of the Dependency Finder toolset [Tes10a] and then compare the XML files of two versions of a class file and report the differences.

4.4 Summary

In this chapter we discussed many of the related works and their applicability issues in our problem domain. Some of them like [Tes10a], [LBL+10] and [RST+04] will be discussed in Chapters 6 and 7.

The next chapter presents an overview of our work processes and also discusses some of the software engineering principles we have followed to make our tools more robust.

Chapter 5

Process Overview

In this chapter, we present a brief overview of finding modifications in class files, and of the dependency graph generation process. We also discuss, briefly, all the tools and their interaction. Later, we discuss different software engineering principles and their application throughout our software development process.

5.1 Major Steps in Our Process

As mentioned in Chapter 2, the *modification finding* and *access dependency graph* generation process are fully automatic without any human intervention. In this section we briefly describe the major steps that we follow in accomplishing these two tasks. For each consecutive step, output of one step will be the input of the next step. We call these major steps because each step may actually consist of internal sub-steps which will be discussed in detail in subsequent chapters.

Figure 5.1 shows the major steps of the modification finding process. More specifically, these are:

- We have the original and modified versions of the set of class files that are associated with a patch. (This information is obtained from the Patch Analyzer tool introduced in Chapter 2).

- The XML Generator uses the *ClassReader* tool of *Dependency Finder*

[Tes10a] toolset to generate one-to-one equivalent XML files of the class files (both original and modified).

- Finally the difference generating tool (Comparer) generates XML files containing difference (modification) information comparing the original and modified versions of the XML files.

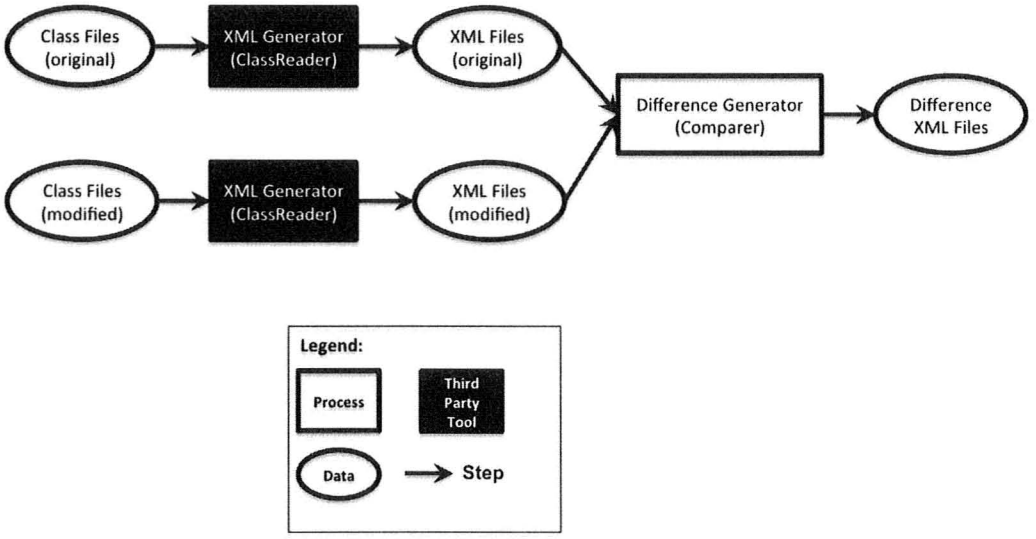


Figure 5.1: Major Steps in Modification Finding

Figure 5.2 shows the major steps of the dependency graph process. More specifically, these are:

- Using the *classpath*[1] information the File Manager detects which directories (containing class, jar, zip files) need to be processed and also unjars and unzips jar and zip files, respectively, to class files.
- The XML Generator use the *ClassReader* tool of *Dependency Finder* [Tes10a] toolset to generate one-to-one equivalent XML files of the class files.
- The Entity Handler uses the XML files to build the entity list, i.e., the list with all the methods and fields, as well as the inheritance information that exists among classes and interfaces.

[1]classpath is the collection of directories where Java runtime looks for classes

- The Dependency Handler uses the same XML files to build the access dependency graph. This is a complex step because it has to handle some of the side issues like dynamic binding.
- The entity list, access dependency graph and inheritance information are all inserted into database tables.

The output from both of these processes, namely, the the XML files containing the modification information and the access dependency graph serves input as input to the Impact Analyzer tool as mentioned in section 2.7.

5.2 Software Engineering Principles

In this section, we discuss some important software engineering principles which are essential to successful software development [GJM03] and their role and impact on our effect analysis project; in particular, the work done in this thesis. Although these principles appear to be strongly related, we prefer to describe them separately and in general terms.

5.2.1 Rigor and Formality

Rigor [GJM03] stands for precision and exactness – which is an intuitive quality and cannot be defined in a rigorous way in software development. Various degrees of rigor can be achieved; the highest among them being formality where the whole software process is driven and evaluated by mathematical laws. The whole software design and development need not be formal (and this is often impossible or very difficult) but we must be able to identify the level of rigor and formality that should be achieved.

In our process, we try to maintain formality by representing the access dependency graph as a set of nodes and edges; which is a typical formal representation of any mathematical relation. Also, we store everything in relational database (Oracle) tables, which is another way of formally storing information.

Rigor and formality also apply to the whole software development process. During programming (a traditional formal approach in the software development process), we directly translate the dependency relationship into the Java

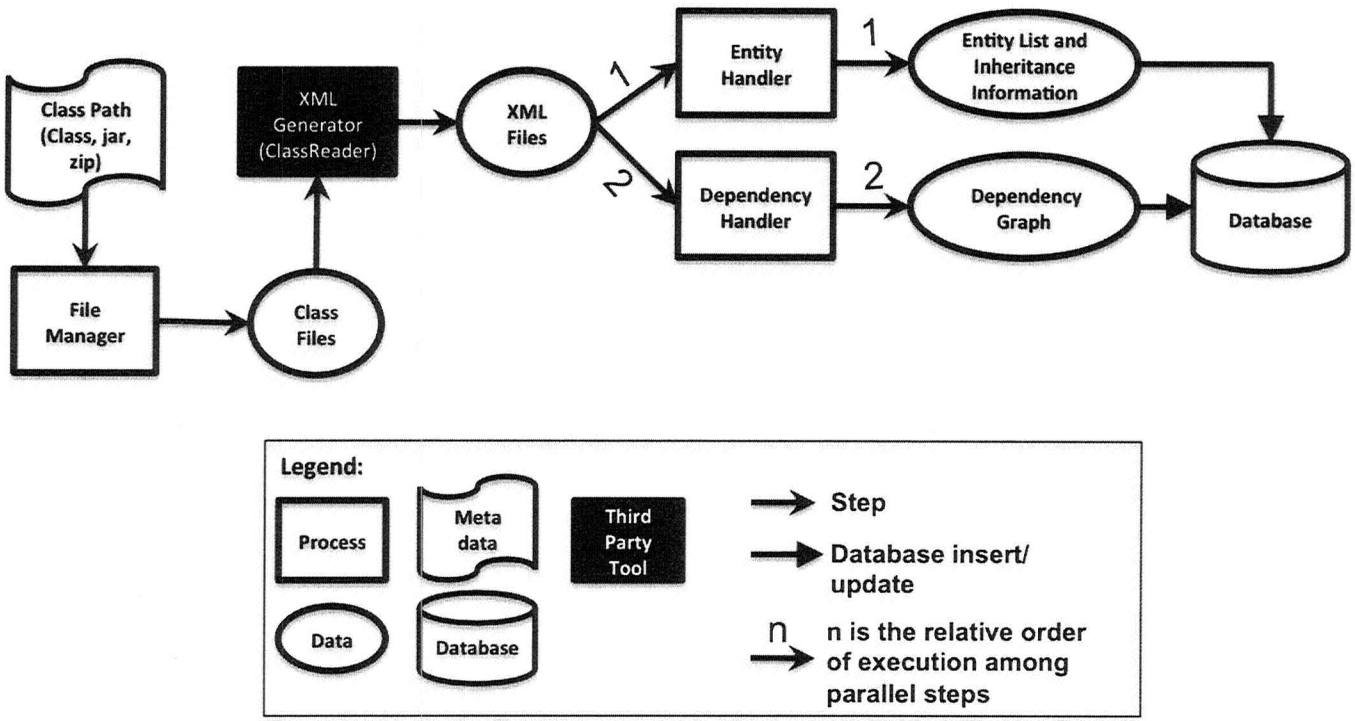


Figure 5.2: Major Steps in Analyzing Dependency

programming objects which are automatically checked and verified for correctness by the Java compiler. Also, as the source of our work and as a way of representing modification between successive versions of class files, we are using XML format, which is a well known formal representation for transferring data from one medium to another.

5.2.2 Separation of Concerns

Separation of concerns helps us deal with different aspects of the problem while concentrating on each independently at a time. Different type of separation of concerns are in practice during software development process [GJM03]. In our case, most of them deal with the higher level design of the impact analysis project tool suite architecture shown in Chapter 2. One important type of separation of concerns is to work with different parts of the problem separately. Using modular development strategy, we have divided the dependency graph generator and modification finder software into several steps. At each step, we are not concerned with the next steps. Necessary adjustments in both the design and the previous steps are made depending on the current step of development. In this way, we can easily concentrate on the current step. For example, while generating XML files from class files, we do not worry about the dependency graph generation process; thus cutting the problem into subproblems.

5.2.3 Modularity

A modular system is a system that is composed of modules. Modularity is essential to build a well structured, layered and maintainable software. As shown in Figure 5.2, we divide our whole process into modules where each module takes care of a different part of the process; thus implementing the principle of separation of concerns. First, the whole process is divided into smaller modules. Then we concentrate on individual module design; following a top down design process.

We use Java language that comes with extensive modularity support with classes and packages. Each major module in our process uses the output of only

a few previous modules; thus indicating low coupling. However, the functions (methods) inside the modules are related strongly to give high cohesion. We keep all the common codes of different modules (in our case, classes) in separate modules (classes) to reduce coupling and also to eliminate repetition of codes from the program.

5.2.4 Abstraction

Abstraction is a basic technique for understanding and analyzing complex problems [GJM03]. By abstraction, we can ignore the complex details of an object and concentrate on the facts that we think relevant. We use Java as our implementation which has a huge collection of abstract data types. Using abstract data types like ArrayList, HashMap we hide much of the internal details of the in memory data storage. This provides us with better program understanding and easily maintainable software.

5.2.5 Anticipation of Change

Software may undergo changes constantly. These changes may be due to elimination of errors or future adaptation in different platforms. Basically, incorporating anticipation of change in the design strategy means to isolate the likely changes in specific portions of the software so that future changes will be restricted to those portions only [GJM03].

We have represented the common data (data that is used by several modules) in one separate Java module (class). We can just make minimum necessary amendments in that module whenever any common data (e.g. a directory path) needs to be changed either for a change in the file system or for use of the software in a different platform. Also, the major tasks are represented through different modules, which ensures that a change or modification can be carried out by making minimum necessary amendments inside one module (class) or a function (method) without drastically changing the API's of individual modules.

Reusability

Reusability is a software quality which is strongly affected by the anticipation of change. The use of Java as the implementation language facilitates reusability of our code. For example, the code of the `XMLGenerator` and `XMLHelper` classes in the modification finder tool was reused in the dependency graph generation tool. Similarly, the code for the `ConnectionManager` class can be reused in any tool where there is a need for creating and closing database connections.

5.2.6 Generality

The principle of generality may be stated as follows: "Every time you are asked to solve a problem, try to focus on the discovery of a more general problem that may be hidden behind the problem at hand" [GJM03]. Although the system we are currently working on is Java based, there is a good probability that we may have to deal with systems based on other languages. So while designing the tool suite, a prime concern was to ensure that different architectures can be represented with minimal changes in the tools. Also another concern was to make the tool suite executable on different platforms. This increases the portability of the tools on different architectures.

Portability on Different Architectures

As far as handling systems based on languages other than Java are concerned, our tools are designed in such a way that they can be handled with little extra effort. For example, we are using XML representation of Java bytecode. Had it been any other language, we could similarly represent the compiled binaries of that language source code in XML representations and then use those in the same fashion. Of course, we might need to write tools for converting the binaries into XML first (which we didn't need in this case, because the existence of some off the shelf bytecode analysis tools). However, because of the modular nature of our design, that extra tool would be easily pluggable into our toolset without violating the original design hierarchy. And the relationship among entities can be represented with Java's abstract data types (just as we

do in our case) anyway.

As for the platform independence issue, Java runs on almost all platforms because of the Java Virtual Machine (JVM); so our tools should be executable on all platforms as well, with the probable changes in some places like the directory naming convention (for UNIX frontslash '/', for Windows backslash '\') etc. Again, due to the modular nature of our tools, these changes take little effort to be incorporated into our tools.

5.2.7 Incrementality

Incrementality applies to a process that proceeds in an incremental fashion [GJM03]. We can add different features of a process in increments. A good software design must incorporate provision to add new features easily. In our dependency graph generation process, our first goal was to find the entity (method, field) list and then we went on finding the dependency graph itself. In future it is possible to add new features (like augmenting the dependency edges with meta information if necessary) to our process.

5.3 Summary

In this chapter, we discussed the overview of our work processes and also discusses some of the software engineering principles we followed in our way of building the tools.

The next chapter is one of the main subject matters of this thesis – the process of finding modifications between two successive versions of a *class* file.

Chapter 6

Finding Modifications in Java Bytecode

In this Chapter, we discuss in detail the way we carry out the modification finding step introduced in Chapter 2. Since in this thesis, we are only concerned with the Java application and not the database, we will only discuss how we detect changes between subsequent versions of Java bytecode (class) files. First we introduce Java bytecodes and possible kinds of modifications. Then we discuss the modification detection process in detail. Finally we discuss the connection with impact analysis of the work discussed in this chapter.

6.1 Java Bytecode

The Java bytecode (*class* file) format has been described in detail in Sun Microsystem's Java Virtual Machine (JVM) Specification [jvm99]. Here we describe in short the parts that are relevant to our analysis. Each *class* file contains the definition of a single class or interface. A *class* file consists of a single `ClassFile` structure[1]:

[1]In these data structures, u1 and u2 represents 2-byte and 4-byte data quantities, respectively

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
```

[jvm99] describes all the individual members of this structure in detail. We describe below, briefly, the specific members that we have used in our analysis:

- **this_class** denotes the name of the class/interface.
- **super_class** denotes the name of the super class of the class/interface under consideration. It is worth mentioning that if a class/interface has no super class explicitly mentioned in the source file code, then the default super class is `java.lang.Object`.
- **interfaces** denotes the set of interfaces that the class implements.
- **acess_flags** is a 16-bit value that contains the various access related information about the class (e.g., whether it is public or private, whether it is actually a class or an interface, whether it is abstract or not etc.)

- **constant_pool** is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the `ClassFile` structure and its substructures.
- **fields** are the set of fields in the class.
- **field_info** contains detailed information about individual fields.
- **methods** are the set of methods in the class.
- **method_info** contains detailed information about individual methods.

The **field_info** and **method_info** are themselves structures [jvm99]:

```
field_info {
     u2 access_flags;
     u2 name_index;
     u2 descriptor_index;
     u2 attributes_count;
     attribute_info attributes[attributes_count];
    }

method_info {
     u2 access_flags;
     u2 name_index;
     u2 descriptor_index;
     u2 attributes_count;
     attribute_info attributes[attributes_count];
    }
```

We describe these below:

- **access_flags** is a 16-bit value describing all the access related properties (e.g. whether it is public, private or protected; static or non-static) of a field or method. For example, in the case of both a field and a method, if bit 1 is set, then the field or method is public.

- **name_index** is an index to the `constant_pool` data structure that would give us the name of the field.
- **descriptor_index** is an index to the `constant\_pool` data structure that would give us several information about the field or method. In case of a field, our desired information is the type of the field. In case of a method, our desired information is the method's signature and return type.
- **attributes** is the data structure that contains other useful information about the fields and methods. Regarding our analysis, we are interested in the code attribute, local variable attribute and exception handler attributes that represent the code (instructions), local variables and a list of handled exceptions, respectively, of a method.

Although all this information is present in the bytecode, the bytecode is in binary format and we need to decode it for extracting this information from it. However, some off the shelf bytecode analysis tools already exists that can give us the XML equivalent of *class* files. The one we use is the *ClassReader* tool of the *Dependency Finder* [Tes10a] toolset. Below we describe the XML representation with an example.

6.2 The XML Representation

The *ClassReader* tool converts the *class* file into a human readable XML format. By parsing this XML, we can get all the required information described in the previous section. Consider the class *TestClass* in Listing 6.1

```
public class TestClass {

    public int i = 1;
    private String str;

    public void test(int b){
        System.out.println(inc() + b);
    }
```

```
    private int inc(){
        return i++;
    }
}
```

Listing 6.1: Sample Class

The XML file generated by *ClassReader* for this class is shown in Listing 6.2, with some of the details hidden as ellipses.

```
<classfile magic-number="0xCAFEBABE" minor-version="0" major-
    version="49" access-flag="00000000 00100001">
    <constant-pool>
        ...
    </constant-pool>
    <public/>
    <super/>
    <this-class>TestClass</this-class>
    <superclass>java.lang.Object</superclass>
    <fields>
        <field-info access-flag="00000000 00000001">
            <public/>
            <name>i</name>
            <type>int</type>
        </field-info>
        <field-info access-flag="00000000 00000010">
            <private/>
            <name>str</name>
            <type>java.lang.String</type>
        </field-info>
    </fields>
    <methods>
        <method-info access-flag="00000000 00000001">
            <public/>
            <name>&lt;init&gt;</name>
            <signature>TestClass()</signature>
            <attributes>
                <code-attribute>
                    <length>10</length>
                    <instructions>
                        ...
                    </instructions>
```

```
                <attributes>
                    ...
                    <local-variable-table-attribute>
                        <local-variable pc="0" length="10"
                            index="0"><name>this</name><
                            type>TestClass</type></local-
                            variable>
                    </local-variable-table-attribute>
                </attributes>
            </code-attribute>
        </attributes>
    </method-info>
    <method-info access-flag="00000000 00000001">
        <public/>
        <name>test</name>
        <return-type>void</return-type>
        <signature>test(int)</signature>
        <attributes>
            <code-attribute>
                <length>13</length>
                <instructions>
                    ...
                </instructions>
                <attributes>
                    ...
                    <local-variable-table-attribute>
                        <local-variable pc="0" length="13"
                            index="0"><name>this</name><
                            type>TestClass</type></local-
                            variable>
                        <local-variable pc="0" length="13"
                            index="1"><name>b</name><type
                            >int</type></local-variable>
                    </local-variable-table-attribute>
                </attributes>
            </code-attribute>
        </attributes>
    </method-info>
    <method-info access-flag="00000000 00000010">
        <private/>
```

```
                <name>inc</name>
                <return-type>int</return-type>
                <signature>inc()</signature>
                <attributes>
                    <code-attribute>
                        <length>12</length>
                        <instructions>
                            ...
                        </instructions>
                        <attributes>
                            ...
                            <local-variable-table-attribute>
                                <local-variable pc="0" length="12"
                                    index="0"><name>this</name><
                                    type>TestClass</type></local-
                                    variable>
                            </local-variable-table-attribute>
                        </attributes>
                    </code-attribute>
                </attributes>
            </method-info>
        </methods>
        <attributes>
        </attributes>
    </classfile>
</classfiles>
```

Listing 6.2: XML File for *TestClass*

Although the XML format is quite messy to look at, it is at least human readable. A careful inspection of the file quickly reveals that this is actually a one-to-one representation of the Java binary bytecode and all the information we could get from the bytecode (details on class, methods, fields) is a also present in the XML which can easily be parsed by any standard XML parser (e.g. DOM parser).

6.3 Types of Changes Between Two Versions of Bytecode

Changes can occur in a number of ways between subsequent versions of a class file. The possible types of changes we figured out are given below:

- **Changes in in the super class.** This may mean deleting an existing super class, or adding a super class to a class that previously didn't have one, or replacing a super class by a new one.
- **Changes in the interfaces the class implements.** This may mean deleting an interface or adding a new one.
- **Changes in the access flag of the class.** This may mean any kind of change in the access related information of that class.
- **Changes in the methods.** This may mean addition of new methods to or deletion of old methods from the class. More importantly, it may also mean any kind of changes in an existing method. This includes:

  1. Changes in the code (instructions) of the method, including local variables and exception handlers.
  2. Change in the return type of the method.
  3. Changes in the access flags (e.g. from public to private) of the method.

  Note that changes in a method do not include a change in the method signature. This is because we consider change in the signature as deletion of a method with the old signature and addition of a method with the new signature.

- **Changes in the fields.** This may mean addition of new fields to or deletion of old fields from the class. It may also mean any kind of changes in an existing field. This includes:

  1. Change in the type of the field.
  2. Change in access flags.

6.4 Some Points to Note

The reader might wonder why we have not included change in a field's initialization in the types of changes listed in the previous section. Also, how are we dealing with the constructors, instance initalizers (which are technically not methods) and static initializers? To understand these issues, it is worth mentioning some points about the *class* file and the types of changes mentioned in the previous two sections. First of all, any constructor, along with the instance initializers in a class would be represented as a method named *<init>* in the *class* file. Even if a class has no explicit constructor mentioned in the source code file, the *<init>* method will be there in the *class* file. If a class has multiple constructors then there is one *<init>* method generated for all of them with different signatures. Thus the issue of constructors and static initializers are safely taken into account by handling the methods in the *class* file.

The initialization of any non-static field is incorporated in the *<init>* method. Likewise, if a class has static fields and static initialization blocks, then all those blocks and static field initializations are encompassed as a single method called *<clinit>* in the *class* file. So, if the initializing value of a field is changed in a subsequent version of a class file, this change is reflected in the *<init>* method (if the field is non-static) and in the *<clinit>* method (if the field is static), but not in the field itself. Only in the case of a change of type or change in the access flags, is the change reflected in the field itself. Thereby the changes in fields initialization are also taken into account by handling the methods in the *class* file.

Another important point is that a method is differentiated from another by its signature, not by just its name, because overloaded versions of methods with different signatures have the same name. So in the case of modification detection, if the signature of a method changes, we consider it as deletion of a method with the old signature and addition of a method with the new signature.

In section 6.6 and in Chapter 7, we will discuss more about the types of changes and their correlation with the dependency graph generation and impact analysis.

6.5 Modification Detection Process

Now that we have described the basic structure of *class* files and also mentioned the possible kinds of changes that might occur between subsequent versions of *class* files, let's get into the details of how we actually detect the modifications. A short overview of the steps of the process was given in Chapter 5. Here we discuss the steps in detail. We have implemented our tool *ClassDiff* in Java for carrying out these steps. Along with what we have done, we also describe which class and which method of our program is used for what functionality. Just to mention, we have two reusable utility classes `ConnecionManager` and `XmlHelper` that helps us managing database connection and creating and writing to XML documents.

6.5.1 Getting Two Versions of a Class File

As shown in Figure 2.4, we get the information about which files changed from the *Patch Analyzer* toolset implemented by Akbar Abdrakhmanov [2]. This toolset saves the necessary information (file name, absolute path etc.) in a database table that has the following columns:

```
Name
Original_Path
Patch_Path
Extension
```

Here `Original_Path` is the path of the original (before patching) file and `Patch_Path` is the path of the modified path (after patching). In our program, we have a class `DatabaseHelper` whose method `getChangedFilePaths` gets us the paths of the original and modified *class* files using a database function `FN_GET_CHANGED_FILES_PATHS`. For storing the information in memory, we use instances of a class `FilePath` that records the file name and the absolute paths of the original and modified files.

[2](in progress)

6.5.2 Common Data Holder

Before we move further, we should mention that for sharing some common data and constants across the programs such that directory names where the XML files would be stored, names of database functions and procedures etc., we have a class `Data`. For example, the directory paths where the corresponding XML files for original and modified versions of the XML files would be placed are stored in the following two fields of that class, respectively:

```
newXmlPathPrefix
oldXmlPathPrefix
```

As the reader can guess, this kind of storage facilitates reusability, like for example, if any path need to be changed, we can just change the value of the corresponding field of this class that holds that path. In Chapter 7, we will see that we use similar kind of common data file for our dependency graph generation process as well.

6.5.3 Converting the Class File into XML Format

Using the *ClassReader* tool of the *Dependency Finder* [Tes10a] toolset we convert both versions of the *class* file to equivalent XML format. We have a class `XMLGenerator` whose method `generateSingleXmlFile` does this job for us.

6.5.4 In-Memory Data Structure For Classes

Having generated the XML equivalent for both versions of a *class* file, we parse the XML files and represent both of them as in-memory data structure representing classes:

```
public class Class {
    String name;
    String superClass;
    ArrayList<String> interfaces;
    String access;
```

```
    HashMap<String,MethodInfo> methods;
    HashMap<String,FieldInfo> fields;
}
```

The members `name`, `superClass` and `access` represents the class's name, super class and access flag, respectively. Since a class can implement more than one interfaces, we use Java's `ArrayList` data type for representing the list of interfaces and. For representing methods, we use `HashMap` where the *key* is the method's signature and the *value* is another data structure `MethodInfo`. Similarly for fields also, we use `HashMap` where the *key* is the field's name and the *value* is another data structure `FieldInfo`. The classes representing these two data structures are below:

```
public class MethodInfo {
    String returnType;
    String access;
    String code;
}

public class FieldInfo {
    String type;
    String access;
}
```

The `access` is the access flag of the method or field. The `returnType` is the return type of the method and `code` represents the collection of instructions, local variables and exception handlers of the method. In the case of a field, `type` represents the type of the field. These explanations are analogous to the ones made in section 6.1 where we discussed the bytecode structure. The in-memory representations of the *class* files are built up using the `buildStructureFromXml` method of class `ClassBuilder` in our program.

Reasons for Using HashMaps

The reader might wonder why we are keeping hashmaps for storing the methods and fields, and not a mere list (like ArrayList). The reasons that we need

a kind of data structure that can map a method signature to its information or a field's name to its information. A list structure wouldn not help us much in this regard. Also by keeping hashmaps, it is very easy to compare method-to-method or field-to-field between subsequent versions of *class* files.

6.5.5 XML Representation of the Modifications

Since the modification information is needed for the impact analysis (which is not in the scope of this thesis, but is a part of the impact analysis project), we have decided to represent the modification information in XML format, so that during impact analysis, it can be easily parsed and the information can be easily extracted.

For building up the XML, we first need to define the XML format. The full DTD (Document Type Definition) file defining the XML format is a part of our proprietary code-base. Here we explain the steps of how we build up the XML and then we demonstrate the process with an example.

Algorithm 6.1 lists a high level representation of our modification detection process. It compares the super class, interfaces, access flags, methods and fields and records any change, addition or deletion of any of these things in the output XML file.

Since Algorithm 6.1 is ata very high level of abstraction, below we describe the process in detail.

The in-memory representation of the two versions of the class files discussed in the previous subsection are compared member-to-member and if there is any difference between the two members, a DOM (Document Object Model) node is created and the difference is recorded in that node, possibly with the help of XML attributes. For example, if the super class of a class A is changed from B to C, the corresponding DOM node will be:

```
<superclass old="B" new ="C">
```

More formally, the steps for building up the XML is as follows:

1. Create an XML Document.

Input: original class file C
modified class file C'
Output: an XML file D
Data: s = superclass of class C
s' = superclass of class C'
I = interfaces of class C
I' = interfaces of class C'
a = access flag of class C
a' = access flag of class C'
M = set of methods of class C
M' = set of methods of class C'
F = set of fields of class C
F' = set of fields of class C'

```
1  begin
2      compare s and s' and record superclass difference in D if s ≠ s';
3      compare I and I' and record any added or deleted interface in D if
       I ≠ I';
4      compare a and a' and record any difference in flags in D if a ≠ a';
5      compare M and M' and record any added, deleted or changed
       method in D if M ≠ M';
6      foreach added method c do
7          record the dependency (to other methods and fields) information
           in D;
8      end
9      foreach changed method c do
10         record the details of changes (return type, access or instructions)
           in D;
11         record the dependency (to other methods and fields) information
           changes in D;
12     end
13     compare F and F' and record any added, deleted or changed field in
       D if F ≠ F';
14     foreach changed field c do
15         record the details of changes (type, access) in D;
16     end
17 end
```

Algorithm 6.1: The High Level Algorithm for Finding Modifications in bytecode

2. Create a DOM node called `<classdiff>` with an attribute `name`. The value of this attribute is the fully qualified name[3] of the class. This node is the *root* element of the XML.

3. Compare the super classes.

   - If the super class of the class has changed, create a DOM node called `<superclass>` with two attributes, `new` having the new super class name as value and `old` having the old super class name as value. Add this node as a child of *root*.

4. Compare the interfaces.

   - For each added interface create a DOM node called `<added>` with the new interface name as text value.
   - For each deleted interface create a DOM node called `<deleted>` with the deleted interface name as text value.
   - If there is any added or deleted interfaces found from the previous two steps then create a DOM node called `interfaces` and add the nodes from the previous two steps as children of this node. Add this node as a child of *root*.

5. Compare the access flags.

   - For each newly set access bit, create a node called `added` with the with the type of access as text value.
   - For each newly unset access bit, create a node called `deleted` with the with the type of access as text value.
   - If there is any newly set or unset flag bit found in the previous two steps, create a DOM node called `access` and add the nodes from the previous two steps as children of this node. Add this node as a child of *root*.

6. Compare the methods.

[3]The fully qualified name of a class is the class name preceded by its package hierarchy separated with dot (.)

- For each deleted method, create a DOM node called `deleted` with the method signature as text value.
- For each newly added method, create a DOM node called `added` with two attributes, `access` having the access flag and `signature` having the signature of the method. Record the dependency information of this method.[4].
  - For each field dependency create a DOM node called `field-dependency` with an attribute `static` whose value will be `yes` if that field is static and `no` if that field is non-static. The text value of this node is the field name.
  - For each method dependency create a DOM node called `method-dependency` whose text value will be that method's signature.
  - If any dependencies were found in the previous two steps, create a DOM node called `dependencies` and those dependency nodes as its children. Add this node as a child of `added`.
- For each method whose signature hasn't changed compare them.
  - If the return type has changed create a DOM node called `returntype` with two attributes, `new` having the new return type and `old` having the old return type.
  - Compare the access flags.
    - For each newly set access bit, create a node called `added` with the with the type of access as text value.
    - For each newly unset access bit, create a node called `deleted` with the with the type of access as text value.
    - If there is any newly set or unset flag bit found in the previous two steps, create a DOM node called `access` and add the nodes from the previous two steps as children of this node.

[4]We do not go into detail here about how we find the dependencies. Chapter 7 will discuss the dependency issues in grater detail

- If the instructions, local variables or exception handlers have changed then create a DOM node called `instructions`. Record the dependency information change as well.
  - For each added method dependency create a DOM node called `addedcall` with that method signature as text value.
  - For each removed method dependency create a DOM node called `removedcall` with that method signature as text value.
  - For each added field dependency create a DOM node called `addedaccess` with that field signature as text value. This node will have an attribute `static` with value `yes` if that field is static and `no` if that field is non-static.
  - For each removed field dependency create a DOM node called `removedaccess` with that field signature as text value. This node will have an attribute `static` with value `yes` if that field is static and `no` if that field is non-static.
  - If any dependency information is found in the previous steps, then create a DOM node called `dependencies` and and those dependency nodes as its child. Add this node as a child of the `instructions` node.
- If any changed information has been found in the previous steps, create a DOM node called `methodinfo` and and all those change information nodes as its child.
- If the `methodinfo` node has been created then create a DOM node called `changed` with an attribute `signature` having the method method signature as value. Add the `methodinfo` node as a child of this node.

7. Compare the fields.
   - For each deleted field, create a DOM node called `deleted` with the field name as text value.
   - For each newly added field, create a DOM node called `added` with three attributes, `access` having the access flag, `name` having the

name of the field and `type` having the type of the field.

- For each field whose name hasn't changed compare them.
  - If the type has changed create a DOM node called `type` with two attributes, `new` having the new type and `old` having the old type.
  - Compare the access flags.
    * For each newly set access bit, create a node called `added` with the with the type of access as text value.
    * For each newly unset access bit, create a node called `deleted` with the with the type of access as text value.
    * If there is any newly set or unset flag bit found in the previous two steps, create a DOM node called `access` and add the nodes from the previous two steps as children of this node.
  - If any changed information has been found in the previous steps, create a DOM node called `fielddinfo` and and all those change information nodes as its child.
  - If the `fieldinfo` node has been created then create a DOM node called `changed` with an attribute `name` having the method method signature as value. Add the `fieldinfo` node as a child of this node.

In our program, we have a class `Comparer` which has different methods for comparing superclass, interfaces, methods, fields for carrying out the above steps. As a driver class of all the Java classes in our program mentioned so far, we have a class `ClassDiff` .

Let's get into the example. Consider the two versions of the class C shown in

```
public class C extends A implements D{// extends B{

    private String str = "hello";
    private int count = 0;
    String place = "hell";
```

```
    public void printMessage(String name){
        System.out.println(str + " " + name);
        System.out.println("Welcome to " + place);
        count++;
    }

    private void changeStr(){
        if(str.equals("hello")){
            str = "hi";
        }
        else{
            str = "hello";
        }
    }

    private void changePlace(){
        if(place.equals("heaven")){
            place = "hell";
        }
        else{
            place = "heaven";
        }
    }

}
```

Listing 6.3: Original Version of Class C

```
public class C extends B implements D, E{// extends B{

    private String str = "hello";
    private short count = 0;
    String place = "heaven";
    int max = 100;

    public void printMessage(String name){
        System.out.println(str + " " + name);
        System.out.println("Welcome to " + place);
        increment();
    }
```

```
    private void increment(){
        count++;
        if(count < max){
            count++;
        }
    }

    private void changeStr(){
        if(str.equals("hello")){
            str = "hi";
        }
        else{
            str = "hello";
        }
    }

    public void changePlace(){
        if(place.equals("heaven")){
            place = "hell";
        }
        else{
            place = "heaven";
        }
    }


}
```

Listing 6.4: Modified Version of Class C

A careful inspection of these two version of Class C reveals the following differences between the two:

1. Change in the super class, A in the original (old) version, B in the modified (new) version.

2. Addition of an extra interface implementation – E.

3. Addition of the new field `max`.

4. Initialization of the new field `max` to the value 100.

5. Change of type of the field `count` from int to short.
6. Change in initialization value of the static field `count`.
7. Change in initialization value of the non-static field `place`.
8. Change inside the `printMessage` method – addition of a call to the `increment` method.
9. Addition of the `increment` method.
10. Change of the private access of method `changePlace` to public.

Let's see how our XML output represents these differences. As described above, both versions of the *class* file (in this example, C.class) are first converted into the XML format generated by *ClassReader.*(We are not showing those class-equivalent XML format here because they won't help too much in understanding this example). Then they are parsed and converted to in-memory data structures. By programmatically comparing the two data structures, we get the XML snippets below (created using DOM) for the modifications (changes) listed above:

Change 1 is handled by the following snippet.

```
<superclass new="B" old="A"/>
```

Change 2 is handled by the following snippet.

```
<interfaces>
        <added>E</added>
</interfaces>
```

Changes 3 and 5 are handled by the following snippet.

```
<fields>
    <changed name="count">
        <fieldinfo>
            <type new="short" old="int"/>
        </fieldinfo>
    </changed>
    <added access="0000000000000000" name="max" type="int"/>
</fields>
```

In fact, any kind of access or type change of the fields would be represented inside the `<fields>` tag. Finally, changes 4, 6, 8, 9 and 10 are handled by the following snippet.

```
<methods>
    <changed signature="printMessage(java.lang.String)">
        <methodinfo>
            <instructions new="new" old="old">
                <dependencies>
                    <addedcall>C:increment()</addedcall>
                    <removedaccess>C:count</removedaccess>
                </dependencies>
            </instructions>
        </methodinfo>
    </changed>
    <changed signature="<clinit>()">
        <methodinfo>
            <instructions new="new" old="old"/>
        </methodinfo>
    </changed>
    <changed signature="C()">
        <methodinfo>
            <instructions new="new" old="old">
                <dependencies>
                    <removedcall>A:A()</removedcall>
                    <addedcall>B:B()</addedcall>
                    <addedaccess static="no">C:max</addedaccess>
                </dependencies>
            </instructions>
        </methodinfo>
    </changed>
    <changed signature="changePlace()">
        <methodinfo>
            <access>
                <added>public</added>
                <deleted>private</deleted>
            </access>
        </methodinfo>
    </changed>
    <added access="0000000000000010" signature="increment()">
        <dependencies>
```

```
            <field-dependency static="no">C:max</field-dependency>
            <field-dependency static="yes">C:count</field-dependency>
        </dependencies>
    </added>
</methods>
```

In fact, any kind of changes related to methods and changes in the fields (static or non-static) initialization would be reflected inside the `<methods>` tag. As mentioned earlier, in Java bytecode, any initialization of a non-static field is incorporated inside the instance initializer *<init>* method, which represents the constructor; and any static initialization is incorporated inside the *<clinit>* method. That's why changes 4 and 7 are enclosed inside the changed method with signature `C()`, which is the constructor, because these changes are the changes in non-static initialization. Similarly change 6 is enclosed inside the changed method with signature `<clinit>()`, because these changes are changes in static initialization.

6.6 The Impact Analysis Connection

In the previous section, we discussed detecting changes between *class* files in general. However, considering the level of abstraction at which our change impact analysis will be carried out, it is important to note that not all kinds of changes are important for us. For the impact analysis, as described in Chapter 2, we need to know what methods or fields have undergone changes, but we are not really interested in knowing the changes at a finer grain, for the time being. For example, in case of a modification inside a method, our change detection process reports what kind of change the method actually encountered – return type, instructions or access. But for our proposed impact analysis, all we need to know is the fact that something was changed in the method. Again, in case of changes in the instructions, our change detection process also reports whether the change includes change in dependency to other methods or fields or not. This information, although not needed in the proposed impact analysis, would be needed in the maintenance of the access dependency graph as will be discussed in Chapter 8.

Another point worth mentioning is that some changes are complementary to others. For example, in the example shown in the previous section, the change in the method `increment` cannot be done without introducing the new field `max`. As another example, if the public access of a method or field is changed to private, that might also indicate subsequent changes in other methods from other classes; because the methods that were previously accessing a public field or method directly, can no longer do that because that method or field has now gone private. In this kind of case where two changes are complementary, the one that comes ahead in a control flow might be sufficient for impact analysis. In Chapter 7, the correlation of changes with the impact analysis and dependency graph generation will be discussed in greater detail.

In general, despite the fact that our proposed impact analysis does not use all the information generated by our modification finding process, keeping in mind that in future we might want to work at a more finer grain, the informative output serves as a good source for use in any next order processing. In addition, from the software engineering viewpoint, it is always a good idea to make a tool more general, and considering future reuse.

6.7 Other Information Related to Modification

In addition to the information related to the change in subsequent versions of a *class* file, there can be new *class* files introduced by a patch, as well as deleted class files, which are also needed for the impact analysis phase. Information about new or deleted files are directly available from the patch analysis and has nothing to do with the modification finding process. This information will be directly used in the impact analysis process.

6.8 Summary

In this chapter, we discussed the process of finding modification between two versions of java bytecodes in detail. This information is one of the inputs that will be used for impact analysis.

The next chapter will discuss another major subject matter of this thesis – building the dependency graph among the entities in the Java environment.

Chapter 7

Building the Access Dependency Graph

In this chapter, we discuss how we build up the access dependency graph among the entities in the Java environment. This work refers to the *Dependency Analyzer (Application)* tool of the toolset shown in Figure 2.4. Before we proceed with details, we need to keep in mind that we have to build the graph in such a way that the impact analysis (which is the next phase in our project toolset) phase can be successfully carried out. So, first we discuss the need for the dependency graph from the impact analysis point of view. We then present the notion of call graph and discuss some factors that affect it which leads us to access dependency analysis. Then we discuss the empirical problems we faced while trying to generate the dependency graph using Soot [LBL+10]. After that, we discuss our complete process of generating the dependency graph. Finally we will get back to how some critical impact analysis issues are handled in our case.

7.1 The Need for the Dependency Graph – Impact Analysis Point of View

Since the ultimate goal of our impact analysis project is to provide the customer with a reduced test suite (and suggest new test cases also if possible),

and finding that reduced test suite depends on a successful impact analysis, we need to keep in mind how the impact analysis will be done while building the access dependency graph.

From our modification finder tool (Chapter 6) we get information about changes among successive versions of *class* files. Along with this, we also have information about added and deleted *class* files directly from the patch analysis tool. In the impact analysis phase, we trace back from the changes (or added entities) to the functions in the customer's application layer to find out which functions there leads to a call-path upto those changed (or added) entities.

Now the big question is: what kind of changes do we want to start our impact analysis from? In [RST+04], as discussed in Chapter 4, Ren *et al.* implemented a tool for change impact analysis in Java. They pointed out 16 kinds of atomic changes shown in Table 7.1.1.

AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of a method
LC	Change virtual method lookup
AF	Add a field
DF	Delete a field
CFI	Change definition of an instance field initializer
CSFI	Change definition of a static field initializer
AI	Add an empty instance initializer
DI	Delete an empty instance initializer
CI	Change definition of an instance initializer
ASI	Add an empty static initializer
DSI	Delete an empty static initializer
CSI	Change definition of a static initializer

Table 7.1.1: Categories of Atomic Changes [RST+04]

Out of these 16 kinds of changes, **AC, DC, AM, DM, CM, AF** and **DF** are directly handled by our modification finder and patch analyzer tool. Since in the bytecode, the instance initializers, instance field initializers and

the constructor are incorporated together in a method called <*init*> , and the static initializers and static field initializers are incorporated in a method called <*clinit*> , our modification finder (along with the patch analyzer) also handles the **CFI, CSFI, AI, DI, CI, ASI, DSI** and **CSI** changes. The only change that is not directly handled by our modification finder tool is the **LC** change, which is the change in virtual method lookup.

First we talk about the **LC** change. According to [RST+04], " **LC** represents changes in dynamic dispatch behaviour that may be caused by various kinds of source code changes (e.g., by the addition of methods, by the addition or deletion of inheritance relations, or by changes to the access control modifiers of methods). **LC** is defined as a set of pairs $\langle C, A : m() \rangle$ indicating that the dynamic dispatch behaviour for a call to $A : m()$ on an object with run-time type C has changed", where C is a sub-class of A. Later on in this Chapter, we will show that we build our dependency graph in such a way that the **LC** changes will be incorporated safely by the impact analysis phase.

Besides those 16 kinds of changes, another kind of change we take into account is the change of type in fields. We will discuss this in detail later on in this chapter when we talk more about including fields in our dependency graph.

Considering everything, we need a graph consisting of nodes and edges that incorporates the dependency relationship among the entities (methods, fields) in the Java environment. This graph structure will be used as a knowledge base during the impact analysis. During the design of our project, our first thought was to build a call graph and use that as a knowledge base. In the next sections, we discuss the notion of call graphs in Java context, factors affecting the call graph, why we chose to switch from call graph analysis to an access dependency analysis and the formal notion of access dependency graph.

7.2 Call Graph in Java Context

Back in Chapter 2, we discussed the concepts of the call graph and access dependency graph in a very short span. Before we move on, we take a look at the notion of call graph in the context of Java.

For any 2 classes A and B, if A's method $a()$ calls B's methods $b_1(), b_2(), \ldots b_n()$ then we consider each of these class-method pairs, $A : a(), B : b_1(), B : b_2(), \ldots B : b_n()$, as nodes of the call graph and the following as edges:

$$
\begin{aligned}
&A : a() \rightarrow B : b_1() \\
&A : a() \rightarrow B : b_2() \\
&\vdots \\
&A : a() \rightarrow B : b_n()
\end{aligned}
$$

where each class-method pair on the left of the arrow is the caller node and the one on the right is the callee node.

7.3 Factors Affecting the Call Graph

In this section we discuss some important factors that affect the call graph especially in an OOP context like Java.

7.3.1 Calling of Methods

According to the Java Virtual Machine Specification [jvm99], a method can be called in 4 ways:

- invokeinterface
- invokespecial
- invokestatic
- invokevirtual

invokeinterface is used to invoke a method declared within a Java interface. For example, consider the Java code:

```
void test(Enumeration enum) {
    boolean x = enum.hasMoreElements();
```

```
    ...
}
```

Here, in the compiled bytecode, *invokeinterface* will be used to call the hasMoreElements() method, since Enumeration is a Java interface, and *hasMoreElements*() is a method declared in that interface. Which particular implementation of *hasMoreElements*() is used will depend on the type of object at runtime.

invokespecial is used in certain special cases to invoke a method Specifically. It is used to invoke:

1. the instance initialization method, <*init*>
2. a private method of the calling class itself
3. a method in a superclass of the calling class

The main use of *invokespecial* is to invoke an object's instance initialization method, <*init*> , during the construction phase for a new object. For example, the following code in Java

```
new StringBuffer()
```

will generate bytecode like the following:

```
...
invokespecial java/lang/StringBuffer/<init>()
```

invokespecial is also used by the Java language by the 'super' keyword to access a superclass's version of a method. For example, in the class:

```
class Example {
       // override equals
       public boolean equals(Object x) {
            // call Object's version of equals
            return super.equals(x);
       }
   }
```

the *super.equals(x)* will generate bytecode like the following:

```
...
invokespecial java/lang/Object/equals(Ljava/lang/Object;)
```

Finally, *invokespecial* is used to invoke a private method, since private methods are only visible to other methods belonging to the same class as the private method.

invokestatic calls a static method (also known as a class method). For example, the code

```
System.exit(1);
```

would generate the following bytecode:

```
invokestatic java/lang/System/exit(1)
```

invokevirtual dispatches a Java method. It is used in Java to invoke all methods except interface methods (which use *invokeinterface*), static methods (which use *invokestatic*), and the few special cases handled by *invokespecial*. For example, consider the following code snippet:

```
Object x;
...
x.equals("hello");
```

will generate bytecode like the following:

```
ldc "hello"
invokevirtual java/lang/Object/equals(Ljava/lang/Object;)
```

The actual method run depends on the runtime type of the object *invokevirtual* is used with. So in the example above, if x is an instance of a class that overrides Object's equal method, then the subclass's overridden version of the equals method will be used.

To handle these four kinds of method call, we have to include in our call graph edges representing calls to methods whenever we encounter any of these four kinds of invocation in the bytecode[1].

[1]Actually, we use the XML equivalent of bytecode, as discussed in Chapter 6

7.3.2 The Dynamic Binding Problem

The dynamic binding problem (also known as the dynamic dispatch or virtual method problem), introduced in Chapter 2 and discussed briefly in the context of some related work, is the process of mapping a message to a specific sequence of code (method) at runtime. This is done to support the cases where the appropriate method cannot be determined at compile-time (i.e. statically) [wik11]. Dynamic dispatch is needed when multiple classes contain different implementations of the same method. This can happen because of class inheritance and interface implementation. Consider Figure 7.1 where class B extends A and C extends B.

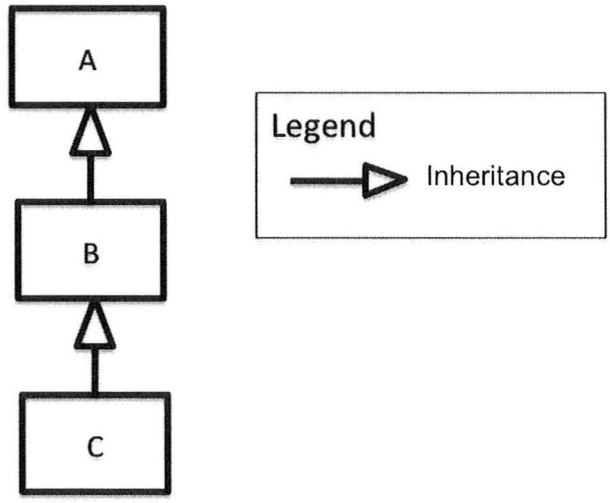


Figure 7.1: Class Inheritance

Suppose that class A has a method $m()$ that is overriden by class B and class C. Now consider the code snippet in Listing 7.1:

```
class A{
        public void m(){
                System.out.println("welcome");
        }
}
class B extends A{
        public void m(){
                System.out.println("hi");
```

```
        }
}
class C extends B{
        public void m(){
                System.out.println("hello");
        }
}
class D{
        public void test(){
                A a = new B();
                a.m();
                a = new C();
                a.m();
        }
}
```

Listing 7.1: Dynamic Binding Example 1

Here, although the static type of a is A, the two calls inside the *test*() method of Class D to $a.m()$ dynamically maps to $B.m()$ and $C.m()$, respectively, although statically both of them are bound to $A.m()$. Notice that if for example, class C didn't override the $m()$ method and if we had code like

```
...
C c = new C();
c.m();
...
```

then the $c.m()$ call would dynamically map to $A.m()$. So in general, this kind of virtual method call can dynamically map to the version of that method in any other class in the inheritance hierarchy, including subclasses and superclasses.

Consider further now Figure 7.2 where interface A is implemented by classes B and C. And both classes implement a method $m()$ from interface A.

Now consider the code snippet in Listing 7.2. The same reasoning applies here too. Although statically both of them are bound to $A.m()$, the two calls inside the *test*() method of Class D to $a.m()$ dynamically maps to $B.m()$ and $C.m()$.

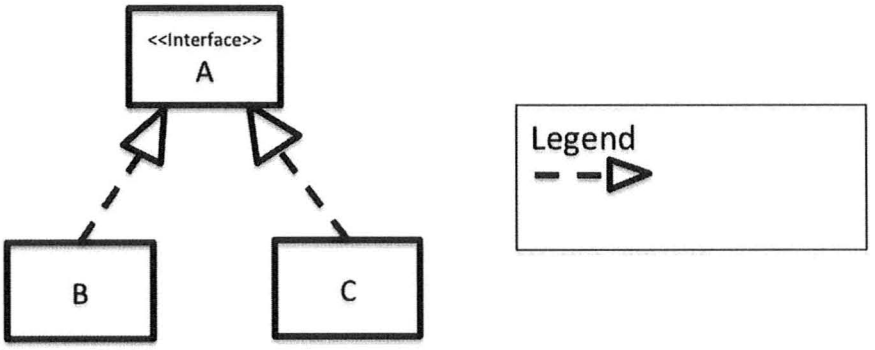


Figure 7.2: Interface Implementation

```
interface A{
        public void m();
}
class B implements A{
        public void m(){
                System.out.println("hi");
        }
}
class C implements A{
        public void m(){
                System.out.println("hello");
        }
}
class D{
        public void test(){
                A a = new B();
                a.m();
                a = new C();
                a.m();
        }
}
```

Listing 7.2: Dynamic Binding Example 2

So class inheritance and interface implementation play similar role in the dynamic binding problem. Let's discuss this issue a bit more with respect to impact analysis with some examples. Consider the scenario in Listing 7.3, where class B extends class A. In the *print* method of class$D$, a is instantiated

as an object of class A first and then B (just to show the effect). Although in each case, a can access methods that are defined in class A only, in the second case a will use the overriden version of the method *test()* defined in class B.

```
class A{
        public void test(){
                System.out.println("hi");
        }
}

class B extends A{
        @Override
        public void test(){
                if(C.check()){
                        System.out.println("hey");
                }
                else{
                        System.out.println("sorry");
                }
        }
}

class C{
        public static boolean check(){
                if(...){
                        return true;
                }
                return false;
        }
}

class D{
        public void print(){
                A a = new A();
                a.test();
                a = new B();
                a.test();
        }
}
```

Listing 7.3: Inheriteance Dependency

Notice that B's version of the *test()* method is dependent on class C's *check()* method, whereas A's is not. If any change happens in C's *check()* method then it will affect B's *test()* (but not A's) and hence D's *print()*. Now in a static analysis, it is hard to determine which version of the *test()* method is being called whenever you encounter a statement like *a.test()*.

For the sake of conservative analysis our first thought was to include all possible calls in the call graph, namely:

$D : print() \rightarrow A : test()$
$D : print() \rightarrow B : test()$

If we include the first one but not the second, we will lose the impact of the change of class C's *check()* method on class D's *print()* method. In reality, there can be many many subclasses (direct or transitive) of A like B. So for the worst case analysis, we have to include all those subclasses in the call graph chain.

Somebody might consider the example in Listing 7.3 a bit foolish. He might wonder why someone would write it like $Aa = newB()$ instead of $Ba = newB()$ when he actually needs an instance of class B. But the problem is in real programs, it might not be known ahead of time whether a is an instance of class A or class B. Below we depict some scenarios that happens in real programs regarding this issue:

7.3.2.1 Scenario 1

Going back to the first example in Listing 7.3, the *print()* method of class D might receive an object of type A from somewhere outside as a parameter, as shown in Listing 7.4. In this case, when analyzing class D, statically there is no way to know whether an instance of class A or of any subclass (direct or transitive) is being used as the parameter. So we cannot say ahead of time which version of the *test()* method will be called.

```
class D{
        public void print(A a){
```

```
                a.test();
        }
}
```

Listing 7.4: Scenario 1

7.3.2.2 Scenario 2

Some standard design patterns directly entail this philosophy of inheritance as a practice of re-usability and encapsulation. Listing 7.5 is a code snippet from one of our master's courses (CAS 703) project (railway simulation) that implements the observer pattern. Class *FilePanel* and *LogPanel* (and a few more actually in our real program) both implement the *Observer* interface. Class *ControlPanel* has a list of Observers (i.e. the classes implementing *Observer*) and whenever its *notifyObservers*() method will get called, it will simply call the *update*() method of all the Observers.

```
interface Observer{
        void update();
}

class FilePanel implements Observer{
        ...
        update(){
                ...
        }
}

class LogPanel implements Observer{
        ...
        update(){
                ...
        }
}

class ControlPanel{
        ...
        ArrayList<Observer> observers = new ArrayList<Observer>();
        ...
        void notifyObservers(){
```

```
                for(Observer observer : observers){
                        observer.update();
                }
        }
}
```

Listing 7.5: Scenario 2

When the *update()* method is called, statically there is no way to know whose update is being called.

It is worth mentioning (and probably clear to the reader by this time) that among the 4 kinds of method invocations discussed in the previous section, only *virtualinvoke* and *interfaceinvoke* are the candidates for the dynamic binding problem, the other two are not.

7.3.2.3 Conservative Analysis

Considering static conservative analysis, our first approach was as follows: for a class (or interface) A and its method $a()$ gets called from class B's method $b()$ and A has classes (or implementations) $A_1, A_2, \ldots, A_n$ in its inheritance hierarchy. Then we included all the following in the call graph:

$B : b() \rightarrow A.a()$
$B : b() \rightarrow A_1.a()$
$B : b() \rightarrow A_2.a()$
$\vdots$
$B : b() \rightarrow A_n.a()$

7.3.2.4 Problems with Conservative Analysis

In the Oracle E-Business suite that we are working on, there are almost 170,000 class files, not considering the jar and zip files. And our first call graph generation program constructed a caller-callee relation with approximately 6 million tuples, without considering the inheritance and implementation. When inheritance and implementation were taken into account, the conservative analysis generated 30-40 million more tuples and the program ran out of memory, even on a 32GB RAM system. Aside from the memory problem, the other problem

is that of those millions of rows, a big percentage may be useless and lead to a huge number of false positives in the impact analysis phase.
Table 7.3.1 shows the top 10 classes (or interfaces) in the Oracle E-Business Suite with the highest number of subclasses (or implementations).

Class or Interface Name	Transitive Subclasses
oracle.jbo.XMLInterface	53,934
oracle.jbo.server.TransactionListener	40,786
oracle.jbo.Properties	36,487
oracle.jbo.VariableManagerOwner	36,458
oracle.jbo.ComponentObject	36,449
oracle.jbo.common.NamedObjectImpl	36,370
oracle.jbo.server.NamedObjectImpl	36,105
oracle.jbo.server.ComponentObjectImpl	36,104
oracle.jbo.server.TransactionPostListener	33,181
oracle.apps.fnd.framework.OAFwkConstants	30,979

Table 7.3.1: Top 10 classes/interfaces with highest number of transitive subclasses/interfaces

In addition to the huge number of subclasses or implementations, another reason for that huge number of tuples was that a method can be called (through *virtualinvoke* or *interfaceinvoke*) from hundreds of places. For example, if there are classes $B_1, B_2 \ldots B_100$ all calling method $m()$ of class A through either *virtualinvoke* or *interfaceinvoke*, then we will have those hundred edges to $A : m()$, plus the number of subclasses multiplied by 100 edges for the conservative analysis.

This huge number of unmanageable and possibly unworthy edges are one of the prime reasons why we switched from a mere call graph analysis to access dependency analysis. We now show how, by using access dependency analysis, we can reduce these huge number of edges dramatically but still keep our analysis static and conservative.

In access dependency analysis, if a method $a()$ of class A calls method $m()$ from class (or interface) B, and classes (and interfaces) $B_1, B_2 \ldots B_n$ are in class B's inheritance (or implementation) hierarchy, then we do not add call edges from $A : a()$ to $B_1 : m(), B_2 : m() \ldots B_n : m()$. Rather we only add a call edge from $A : a()$ to $B : m()$. To handle the dynamic binding statically, we

add edges from $B : m()$ to the $m()$ method of only those transitive subclasses (or implementations) of B who overrode the $m()$ method. Also, if B itself did not override method $m()$, we add an edges from $B : m()$ to the $m()$ method of the closest transitive superclass of B that has $m()$ defined. We explain this with the example in Figure 7.3.

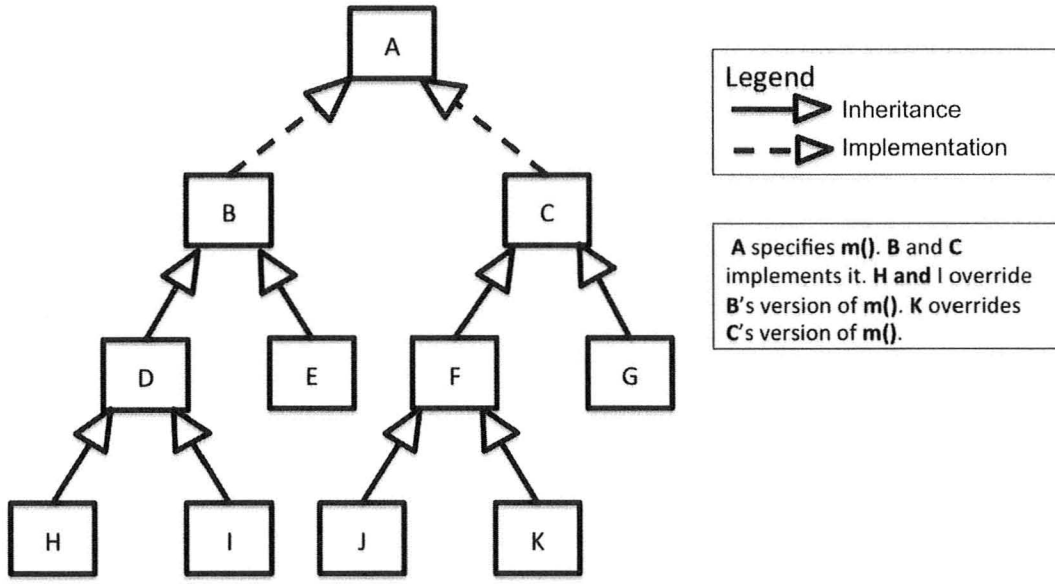


Figure 7.3: Inheritance and Implementation Hierarchy

Here, we show a sample inheritance and implementation hierarchy. Class B and C implements interface A. Classes D and E extend class B; and classes H and I extend class D. On the other side, classes F and G extend class C; and classes J and K extends class F. Consider that interface A specifies one method $m()$ that classes B and C implements. Classes D, E, F, G and J don't override the $m()$ method. Classes H and I override B's version of $m()$ and K override C's version of $m()$.

Now consider there are 10 classes $X_1, X_2 \ldots X_10$, each of which has a method *test*() that calls $A : m()$ with an *interfaceinvoke*. If we had stuck to our conservative analysis then it would have generated all of the following 110 (not all edges shown) call edges:
$X_1 : test() \rightarrow A : m()$
$X_1 : test() \rightarrow B : m()$

$X_1 : test() \rightarrow C : m()$
$X_1 : test() \rightarrow D : m()$
$X_1 : test() \rightarrow E : m()$
$X_1 : test() \rightarrow F : m()$
$X_1 : test() \rightarrow G : m()$
$X_1 : test() \rightarrow H : m()$
$X_1 : test() \rightarrow I : m()$
$X_1 : test() \rightarrow J : m()$
$X_1 : test() \rightarrow K : m()$
$X_2 : test() \rightarrow A : m()$
$X_2 : test() \rightarrow B : m()$
$X_2 : test() \rightarrow C : m()$
$X_2 : test() \rightarrow D : m()$
$X_2 : test() \rightarrow E : m()$
$X_2 : test() \rightarrow F : m()$
$X_2 : test() \rightarrow G : m()$
$X_2 : test() \rightarrow H : m()$
$X_2 : test() \rightarrow I : m()$
$X_2 : test() \rightarrow J : m()$
$X_2 : test() \rightarrow K : m()$
$\vdots$
$\vdots$
$X_10 : test() \rightarrow A : m()$
$X_10 : test() \rightarrow B : m()$
$X_10 : test() \rightarrow C : m()$
$X_10 : test() \rightarrow D : m()$
$X_10 : test() \rightarrow E : m()$
$X_10 : test() \rightarrow F : m()$
$X_10 : test() \rightarrow G : m()$
$X_10 : test() \rightarrow H : m()$
$X_10 : test() \rightarrow I : m()$
$X_10 : test() \rightarrow J : m()$
$X_10 : test() \rightarrow K : m()$

On the other hand, if we use access dependency analysis described above, we only have the following 15 edges:

$X_1 : test() \rightarrow A : m()$
$X_2 : test() \rightarrow A : m()$
$X_3 : test() \rightarrow A : m()$
$X_4 : test() \rightarrow A : m()$
$X_5 : test() \rightarrow A : m()$
$X_6 : test() \rightarrow A : m()$
$X_7 : test() \rightarrow A : m()$
$X_8 : test() \rightarrow A : m()$
$X_9 : test() \rightarrow A : m()$
$X_10 : test() \rightarrow A : m()$
$A : m() \rightarrow B : m()$
$A : m() \rightarrow C : m()$
$A : m() \rightarrow H : m()$
$A : m() \rightarrow I : m()$
$A : m() \rightarrow K : m()$

The first 10 edges are due to the *invokevirtual* invoked to $A : m()$. And the rest 5 are the edges from $A : m()$ to the $m()$ method of only those transitive subclasses or implementations of A that overrode or implemented their own version of $m()$, namely, B, C, H, I and K. Thus, we have a great deal of reduction in the number of edges. Notice that, a similar explanation exists in case there was a *virtualinvoke* from anywhere to the $m()$ method of B or C. Notice also that for example, if there is a *virtualinvoke* from $X_1 : test()$ to $J : m()$, then by the conservative analysis we will have the following 4 edges:

$X_1 : test() \rightarrow J : m()$
$X_1 : test() \rightarrow F : m()$
$X_1 : test() \rightarrow C : m()$
$X_1 : test() \rightarrow A : m()$

whilst the access dependency analysis will give us only 2 edges:
$X_1 : test() \rightarrow J : m()$

$J : m() \rightarrow C : m()$

because C is the closest transitive superclass of J that has the body of method $m()$ defined.

Now considering impact analysis, statically we have all the information in the access dependency analysis as we would have had in the fully conservative analysis. The impact analysis can still be successfully carried out. Consider a change caused by a patch inside the $K : m()$ method, and assume we have any of the following two code segments:

```
class X{
...
        public void test(A a){  // a is actually an instance of class J
                        a.m(); // an interfaceinvoke to A:m()
        }
}

class X{
...
        public void test(C c){  // c is actually an instance of class J
                        c.m(); // a virtualinvoke to C:m()
        }
}
```

With the following edges generated by our access dependency analysis, it is still possible to trace back from the changed method $J : m()$ to $X : test()$

$C : m() \rightarrow J : m()$

$A : m() \rightarrow C : m()$

$X : test() \rightarrow A : m()$

in case of the *interfaceinvoke* and

$C : m() \rightarrow C : m()$

$X : test() \rightarrow C : m()$

in case of the *virtualinvoke*.

Notice that one of the reasons we don't call it a call graph any more is that the edges that we generate to handle the dynamic binding problem statically are not really call edges. For instance, in the above example, $A : m()$ doesn't call $C : m()$ and $C : m()$ doesn't call $J : m()$. What this means is that

a call to $A : m()$ might actually result in call to $C : m()$ or a call to $C : m()$ might actually result in call to $J : m()$. And adding the edges this way (rather than adding edges directly from the original callers) dramatically reduces the number of edges. For example, in the Oracle E-Business suite, as shown in Table 7.3.1, the interface `oracle.jbo.XMLInterface` has over 50,000 transitive implementations or subclasses. But only a few hundred of them had their own body of certain implemented or overridden methods defined in themselves. So the number of edges were reduced to a huge extent after the access dependency analysis was taken into account. It also eliminated our *out of memory* problem altogether.

7.3.3 Including Fields

While initially thinking about the construction of a call graph, we did not think about fields. Under ideal condition, where at least the syntactic validity of the whole program is maintained even after a patch is applied, we do not need to include the fields at all. Considering impact analysis, a change in a field can mean change in the field's initialization, a change in the field's type or a change in the field's access flags.

Now, when a change occurs in a field initialization, this change is not reflected in the *fields* section of the bytecode (or the XML representation of the bytecode we are using). The fields section remains unchanged. It is reflected as a change in the $<init>$ method in case of a non-static field and in the $<clinit>$ method in the case of a static field; which is quite justified. So with respect to impact analysis, to capture a change in a field initialization, it seems that we do not need a calling edge from the method that uses that field to that field itself, because that edge would be useless. Rather it makes more sense to have a calling edge from the calling method to the $<init>$ method of the class which the field belongs to (in case of a non-static field) or to the $<clinit>$ method (in case of a static field). However, careful thinking reveals that, in case of a non-static field, we do not need the calling edge either. Because up above the control flow an instance of that class must have been instantiated somewhere using *invokespecial* and we have the calling edge to that class's $<init>$ method there. Of course, in the case of a static field, we

need an edge to that class's <*clinit*> method because <*clinit*> does not get called explicitly. But in neither case, in the ideal condition, do we need an edge to the field itself.

Now the other kind of change that might occur pertaining to field is change in field type (like long to short) or a change in the access flags (like public to private). Once again, in an ideal case, these changes are bound to be accompanied by subsequent changes in one or more methods. Consider the code in **Listing listing:7.6**.

```
class A{
        public long i = 4;

        public void test(){
                i = getValue();
        }

        private long getValue(){
                ...
                return value; // value is some short value
        }
}

class B{
        public void check(){
                A a = new A();
                long d = a.i;
                ...
        }
}
```

Listing 7.6: Code: Assigning a short value

Here if, the type of the field i is changed from long to short, the return type of the method *getValue* is doomed to change from long to short also. Otherwise there is syntactic invalidity left in the program. Similarly, consider if the access modifier of i is changed from public to private, then class B's method *check*() cannot access i directly as it is doing in the code. There has to be subsequent changes inside class B and class A also if class B wants access to the value of i (like adding a public method in class A giving class B indirect

read-only access to class B). So in these cases also, we do not need the edges to fields under ideal circumstances.

However, keeping in mind that a customer's system actually has the legacy application layer on top of the E-Business Suite, a change in a field type or access flags might leave syntactic inconsistency in the system. The point here is that Oracle patches only modify the E-Business Suite (and the database also, but that is out of the context). So any change inside the E-Business Suite that can lead to a syntactic invalidity will be compensated by other changes elsewhere in the E-Business Suite. This is ensured by Oracle themselves. But considering the legacy nature of the customer's application layer, the interface between the application layer and the E-Business Suite might not be well defined[2]. For example, in the above example, if class B is in the customer's layer rather than the E-Business Suite, the change of access of field i from public to private would lead to syntactic invalidity. For this reason, as a safe conservative approach, we have decided to keep the edges to the fields.

Since accessing a field is not a call, we do not name our graph a *call* graph. This is one of the reasons we call it *access dependency graph*, or just dependency graph in short (The other reason is due to how we handle the huge number of edges due to the dynamic binding problem and will be discussed in section 7.4). And instead of the terms 'caller' and 'callee', we use 'accessor' and 'accessee'. Formally, for any 2 classes A and B (A and B can possibly be the same class), if A's method *a()* accesses B's fields $b_1, b_2, \ldots b_n$ then we consider each of these class-field pairs, $A : a(), B : b_1, B : b_2, \ldots B : b_n$, as nodes of the access dependency graph and the following as edges:

$$
\begin{aligned}
&A : a() \rightarrow B : b_1 \\
&A : a() \rightarrow B : b_2 \\
&\vdots \\
&A : a() \rightarrow B : b_n
\end{aligned}
$$

where each class-method pair on the left of the arrow is the accessor node and

[2]At the time the impact analysis project is being conducted, we don't have a very good knowledge on the customer's legacy layer and its the interface of the E-Business suite to it

the one on the right is the accessee node.

We can note that in Java bytecode, accessing a non-static field is expressed by the instructions *putfield* (write) and *getfield* (read) while accessing a static field is expressed by the instructions *putstatic* (write) and *getstatic*.

Now that we have discussed the issues affecting the call graph, and discussed the reasons behind our switching from a mere call graph to an access dependency graph, we present the details of the access dependency analysis.

7.4 Access Dependency Analysis

Considering everything we have discussed so far, we will formally describe our concepts of the access dependency graph in this section. Below are the criteria we take into account while building our dependency graph using access dependency analysis:

1. For any two classes A and B (where A and B could possibly be the same class, or B may be an interface), if A's method $a()$ calls B's method $b()$ using any of *invokeinterface, invokestatic, invokespecial* and *invokevirtual*, then we add the following edge to the dependency graph:

   $A : a() \rightarrow B : b()$

2. For any two classes A and B (where A and B could possibly be the same class, or B may be an interface), if A's method $a()$ calls B's method $b()$ using either of *invokeinterface* or *invokevirtual*, and B has transitive subclasses or implementations $B_1, B_2 \ldots B_n$ explicitly implementing or overriding $b()$ as its own version, then add the following edges to the dependency graph in addition to the edge described in criteria 1:

   $B : b() \rightarrow B_1 : b()$
   $B : b() \rightarrow B_2 : b()$
   $\vdots$
   $B : b() \rightarrow B_n : b()$

In addition, if B is a class that inherited method $b()$ from some other class but doesn't override $b()$ itself, then add the following edge to the dependency graph:

$B : b() \rightarrow S : b()$
where S is the closest transitive superclass of B up the inheritance hierarchy.

3. For any two classes A and B (where A and B could possibly be the same class), if A's method $a()$ accesses B's field b using any of *putfield, getfield, putstatic* and *getstatic*, then add the following edge to the dependency graph:

$A : a() \rightarrow B : b$

In addition, if b is a static field, also add the following edge to the dependency graph:

$A : a() \rightarrow B :< clinit > ()$

where $<clinit>$ is the bytecode method representing the static initializers of the class.

7.4.1 Sensible Transitive Closure

Notice that every sensible path that could have been traversed by the conservative analysis is also traversable by the access dependency analysis, but possibly with several orders of magnitude fewer number of edges. By sensible, we mean a path that is worth traversing (for example, if a class does not override a certain method, it is not sensible to traverse a path to that method of that class). This means that the transitive closure of the sensible accessor-accessee relation pairs are same for the conservative analysis and the access dependency analysis. So we have the same sensible reachability in the access dependency analysis as we would have had in the conservative analysis.

7.4.2 Handling LC Changes

Its worth noting that our access dependency analysis safely incorporates the **LC** changes [RST+04] mentioned earlier in section 7.1. Since our access dependency analysis adds edges to a the methods of subclasses or implementations of a class defining their own version in case of an *interfaceinvoke* and *virtualinvoke*, we cannot possibly miss any impact related to change in a virtual method lookup. For example, consider the example program taken from [RST+04]. Here the changes (added code fragments) have been shown in boxes.

```
class A {
  public A(){ }
  public void foo(){ }
  public int x;
}
class B extends A {
  public B(){ }
  public void foo(){ B.bar(); }
  public static void bar(){ y = 17; }
  public static int y;
}
class C extends A {
  public C(){ }
  public void foo(){ x = 18; }
  public void baz(){ z = 19; }
  public int z;
}
```

```
class Tests {
  public static void test1(){
    A a = new A();
    a.foo();
  }
  public static void test2(){
    A a = new B();
    a.foo();
  }
  public static void test3(){
    A a = new C();
    a.foo();
  }
}
```

Figure 7.4: Example program. Added code fragments are shown in boxes. [RST+04]

Notice that class C did not override method $foo()$ in the old version. But it does override in the new version. So the $test3()$ method of class $Tests$ now binds the call $a.foo()$ to $C : foo()$, whereas previously it used to bind it to $A : foo()$. This change will be captured in our analysis because we will have the following edges in our access dependency graph of the new program (with others, of course):

$Tests : test3() \rightarrow A : foo()$
$A : foo() \rightarrow B : foo()$
$A : foo() \rightarrow C : foo()$

So it is easy to track back from $C : foo()$ to $Tests : test3()$ following these edges.

So far, we have discussed the factors affecting the call graph, our reasons for switching from call graph to access dependency graph and the formal description of the access dependency graph. We now discuss our full empirical process of generating the dependency graph. For the process, we use the XML equivalent of Java *class* files, just like we did in Chapter 6. Before we move on, we will first discuss our first attempt of generating dependency graph using Soot [LBL+10]. Although we did not use Soot finally in our process, its worth discussing the empirical problems we faced while using it, because during our early project work, we stuck to it for almost two months until we decided to abandon it because of those empirical problems.

7.5 First Attempt to Build Dependency Graph Using Soot

Soot was introduced in Chapter 4. It is a widely used Java optimization framework for optimizing bytecodes and carrying out several kinds of analysis like control flow analysis, data flow analysis, extracting call graphs etc. We attempted to generate control flow graphs and call graphs with Soot. Below we discuss these.

7.5.1 Control Flow Graph With Soot

Soot provides provisions for generating several control flow graphs in its package `soot.toolkits.graph`. Before trying Soot on our E-Business Suite, we tried Soot on some sample small-size projects having order of a few hundred classes on our personal machines (Mac OS X, 2.13 GHz Intel Core 2 Duo, 4 GB SDRAM). However, it turned out that even for a single class, with no

maximum heap size specified, Soot ran out of memory and with 500 MB maximum heap size specified, Soot took almost 40-50 seconds to generate control flows. When we ran Soot in whole program mode to generate control flows for the whole program, it continued to run out of memory and seemed to run forever.

Now compared to the size of our E-Business Suite, our sample projects were pretty small. That's one of the reasons we decided not to do a detailed control flow or a data flow analysis. We switched to generating call graphs with Soot.

7.5.2 Call Graphs With Soot

First we discuss the techniques that Soot uses to generate call graphs and then we discuss our experience applying them. Vijay *et. al* discussed 3 kinds of analyses (introduced in Chapter 4) to generate call graphs which have been incorporated in Soot. We describe these techniques in short here:

Class Hierarchy Analysis (CHA)

Class hierarchy analysis is a standard method for conservatively estimating the run-time types of receivers[3] [BS96]. Given a receiver o of with a declared type d, $hierarchy\_types(d)$ for Java is defined as follows:

- If receiver o has a declared class type C, the possible run-time types of o, $hierarchy\_types(C)$, includes C plus all subclasses of C.
- If receiver o has a declared interface type I, the possible run-time types of o, $hierarchy\_types(I)$, includes: (1) the set of all classes that implement I or implement a subinterface of I, which they call $implements(I)$, plus (2) all subclasses of $implements(I)$.

Its worth noting that this analysis is almost same as the conservative analysis we discussed before in the previous section, except that CHA takes into account only the subclasses and subinterfaces, but not the superclasses.

This analysis results in the call graph with the maximum number of edges.

[3]receiver here is the object on which the method in invoked

Rapid Type Analysis (RTA)

Rapid type analysis [SHR+00, BS96] is a very simple way of improving the estimate of the types of receivers. The observation is that a receiver can only have a type of an object that has been instantiated via a new. Thus, one can collect the set of object types instantiated in the program P, call this *instantiated_types*(P). Given a receiver o with declared type C with respect to program P, they use *rapid_types*(C, P) = *hierarchy_types*(C) $\cap$ *instantiated_types*(P) as a better estimate of the runtime types for o. This particular version of rapid type analysis is called pessimistic rapid type analysis [SHR+00] since it starts with the complete conservative call graph built by CHA and looks for all instantiations in methods in that call graph. The original approach suggested by Bacon and Sweeney [BS96] is optimistic rapid type analysis. In the optimistic approach the call graph is iteratively created, and only instantiations in methods already in the call graph are considered as possible set for computing *instantiated_types*(P).

Variable-type Analysis and Declaration-type Analysis

According to [SHR+00], Rapid type analysis can be considered to be a very coarse grain mechanism for approximating which types reach a receiver of a method invocation. In effect, rapid type analysis says that a type A reaches a receiver o if there is an instantiation of an object of type A (i.e. an expression $newA()$) anywhere in the program, and A is a plausible type for o using class hierarchy analysis.

For a better approach, they point out that for a type A to reach a receiver o there must be some execution path through the program which starts with a call of a constructor of the form $v = newA()$ followed by some chain of assignments of the form $x_1 = v; x_2 = x_1; \ldots; xn = x_{n-1}; o = x_n$. The individual assignments may be regular assignment statements, or the implicit assignments performed at method invocations and method returns. They propose two flow-insensitive approximations of this reaching types property. Both analyses proceed by: (1) building a type propagation graph where nodes represent variables, and each edge $a \rightarrow b$ represents an assignment of the form

$b = a$, (2) initializing reaching type information generated by assignments of the form $b = newA()$ (i.e. the node associated with b is initialized with the type A) and, (3) propagating type information along directed edges corresponding to chains of assignments. These two are Variable-type Analysis and Declaration-type Analysis. The details of these techniques have been discussed in [SHR+00].

With Soot, call graphs can be generated using these different techniques, the computational complexity being the least in case of class hierarchy analysis and the most in the case of variable type analysis.

We tried the class hierarchy analysis to extract the call graph from a single sample *class* file on our local machines and it took almost 50 seconds to generate the call graph, the maximum heap size being specified as 800 MB. Without specifying the heap size, it was still running out of memory. Then we went on and tried to generate the call graph for our sample small size projects in whole program mode and even with 2 GB of maximum heap size specified, it ran out of memory. Considering this performance, the call graph generation process of Soot didn't seem feasible in case of our huge sized E-Business Suite at all.

The amount of time needed by Soot is due to the fact that when it begins an analysis from a particular class, it loads that class into memory and then subsequently loads all the classes that is directly or transitively referenced by that class, in addition to carrying out all the computations. And thereby, when executed, it also needs all those classes to be present in its classpath.

Going back to our experience with Soot, we then switched to another approach, that is, using the XML representation of the bytecodes generated by Soot. This XML is rather an operational semantics level of a *class* file (and so much more verbose than the XML generated by the *ClassReader* tool of the *Dependency Finder* toolset). Our intent was to generate the XMLs first and then parse them to extract the dependency information (like *invokeinterface, invokespecial, invokevirtual, invokestatic, getstatic, putstatic, getfield, putfield*).

The XML way worked much better in terms of time and memory. It took approximately 4 seconds, on the average to generate one XML file from a *class*

file. And for our small size test projects, it worked reasonably well.

But then we went on and tried to use the XML way on the machine (Linux 64-bit, 32 GB SDRAM). With individual XML file generation, running out of memory on this machine was not really an issue, but at an average of 4-5 seconds per XML generation, the total time for generating XML files for all the E-Business Suite files would have been approximately one and a half weeks !! In practice, it turned out to be even worse – we ran our program and after 3 days, it was only able to generate XML files for 23,000 *class* files. This is because for some files, the XML generation took more than 10 seconds and because of the possible huge access dependencies among the classes, the loading of hundreds of classes was making it even worse.

As a result, despite being an excellent tool and incorporating excellent techniques like class hierarchy analysis, rapid type analysis, variable type analysis etc., Soot was empirically unable to prevail in our specific problem domain.

Fortunately, soon after these experiences, we came across the *Dependency Finder* toolset [Tes10a] which is actually a suite of tools for analyzing Java bytecode. As mentioned in Chapter 3, among its tool suite, two were of special interest to us – *Dependency Extractor* and *ClassReader*. *Dependency Extractor* generates XML containing information specifically pertaining to dependencies, but lacks some useful information like inheritance, invocation type (*interfaceinvoke, virtualinvoke, specialinvoke, staticinvoke*), field access type (*getfield, putfield, getstatic, putstatic*) etc. So we decided to use *ClassReader* which generates rather an equivalent one-to-one representation of the bytecode containing every information we need.

We have now reached a stage where we can begin describing our dependency graph generation process in detail.

7.6 Graph vs Relation

Just to make a note, in the discussion following, we will use the words *graph* and *relation*, *edge* and *pair*, *entity* and *node* interchangeably. This is because we can imagine the edges of a graph as pairs of a binary relation; and the nodes of a graph as elements of the set on which the relation is built on. For

example, if we have the following call edges: $A : a() \rightarrow B : b()$
$A : a() \rightarrow C : c()$

we can imagine it as a relation having nodes $\{A : a(), B : b(), C : c()\}$ and edges $(A : a(), B : b()), (A : a() \rightarrow C : c())$. Since we will ultimately store the dependency information in relational database tables, technically it will be a relation. But conceptually it is also a graph.

7.7 Access Dependency Graph Generation

In this section we describe the access dependency graph generation process. This process corresponds to the steps shown in Figure 5.2. We have implemented our automated tool *Dependecy Analyzer* in Java for carrying out these steps. With the process detail, we will also describe which tasks are carried out by which class and method in our program. To start with the process we need a classpath from where we can pick up our *class* files.

7.7.1 The ClassPath

From one of the configuration files of the E-Business Suite (courtesy: Akbar Abdrakhmanov), the following classpath was found:

```
/u01/oracle/VIS/apps/apps_st/comn/java/classes,
/u01/oracle/VIS/apps/tech_st/10.1.3/appsutil/jdk/lib/dt.jar,
/u01/oracle/VIS/apps/tech_st/10.1.3/appsutil/jdk/lib/tools.jar,
/u01/oracle/VIS/apps/tech_st/10.1.3/appsutil/jdk/jre/lib/rt.jar,
/u01/oracle/VIS/apps/apps_st/comn/java/lib/appsborg.zip,
/u01/oracle/VIS/apps/tech_st/10.1.2/forms/java,
/u01/oracle/VIS/apps/tech_st/10.1.2/forms/java/frmall.jar,
/u01/oracle/VIS/apps/tech_st/10.1.2/jlib/ewt3.jar,
/u01/oracle/VIS/apps/tech_st/10.1.2/j2ee/OC4J_BI_Forms/applicati-
ons/formsapp/formsweb/WEB-INF/lib/frmsrv.jar
```

Notice that some of the directories leads to jar and zip files. Along with those, some of the other directories of the classpath also have lots of jar and zip

files containing class files in them. At this stage there were some interesting thoughts about these jars and zips going on in our mind. Before we move further with the dependency graph generation process, we will discuss those thoughts.

Influence of Jar and Zip Files

In a good number of patches inspected by us, no jar or zip files were found. Presumably the reason is that the class files inside the jar files are more stable, that is, less likely to change over time and that's why they are jarred or zipped. Despite their stableness, we can't ignore calls that go to jars and zips. Consider Figure 7.5, where there is call from some method in a class A to some method inside some class in jar J_1. That call then followed by some subsequent calls inside other jars (possibly the same one) and finally there is a call back from jar J_n to some method in class B. So unless we record this full call chain, we miss the transitive dependency from A to B.

Moreover, many of these jar files are from the java standard class library (e.g. rt.jar). And since some of Oracle's E-Business Suite's classes might have inherited from them we need to record those for tackling the dynamic binding problem anyway. Finally, we estimated that there are 170,000 thousand class files in the classpath that are not inside any jar or zip file, and 205 jar and zip files, extracting to almost 60,000 class files. Summing up, we get almost 230,000 class files. So incorporating these jars and zips does not introduce too much overhead to our computation considering the overall size.

7.7.2 The High Level Algorithm

Algorithm 7.1 shows a high level algorithm for our dependency graph generation. It is basically an algorithmic re-phrasal of Figure 5.2. First, we use the classpath information to generate XML files from class files; and in case of jar and zip files, extract them and then generate the XMLs. Parsing those XML files we get the list of entities (methods, fields) in classes and assign each entity a unique id (this will be explained later). We also build up the dependency relationship (method call, field access) and augment it with extra

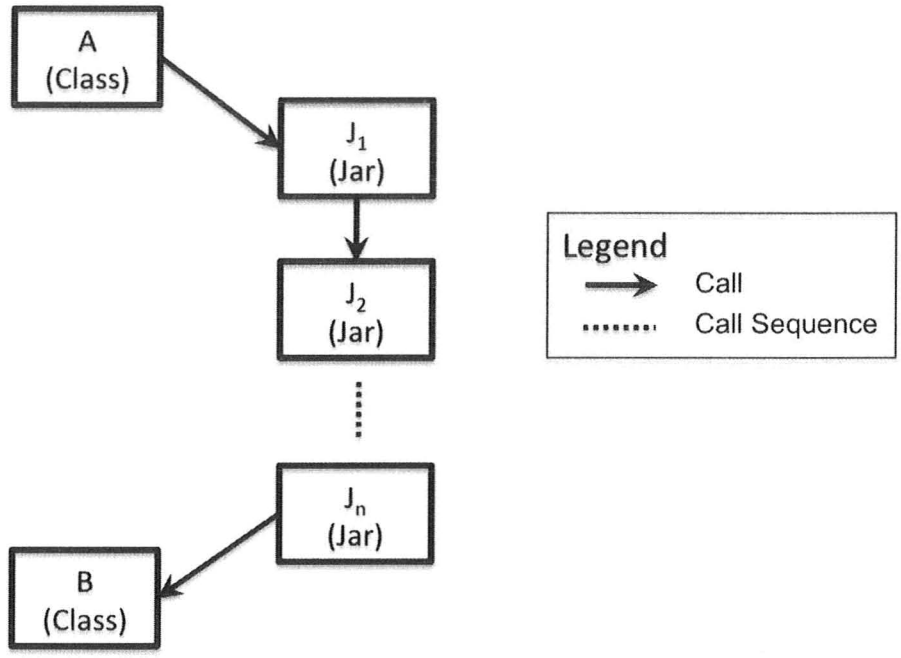


Figure 7.5: Call to and call back from jars

relation pairs arising for sovling dynamic binding issues as discussed earlier. Finally we insert all these information into database for future reuse. In the next few sections, we describe all these steps in detail.

7.7.3 Common Data File

Just like the modification finding process in Chapter 6, here also, we we maintain a class called `Data` throughout our process that holds common data and constants shred by all other classes (modules). These include:

- `inTable` – a hashmap for storing inheritance information
- `entities` – a hashmap for for storing entity (method:field) information
- `nonOverridenPairs` – a hashmap for storing information about methods that were not overriden or implemented by the sorresponding class

Input: classpath CP
C
XML Repository (a directory) R
Output: an XML file D
Data: E = in-memory entity repository
I = in-memory inheritance repository
D = in-memory dependency repository
DB_E = database entity repository
DB_I = database inheritance repository
DB_D = database dependency repository

```
1  begin
2      foreach path p ∈ CP do
3          generate XML files from class files in p and store them in R;
4          extract jar and zip files in p to class files and generate XML files
           from them and store them in R;
5      end
6      foreach XML file x ∈ R do
7          parse the XML file;
8          record all entity information (method, field) in E with unique
           id's for each entity;
9          record inheritance information in I;
10     end
11     foreach XML file x ∈ R do
12         parse the XML file;
13         record all method call information (method, field) in D in terms
           of id's;
14         for virtualinvoke and interfaceinvoke calls add extra information
           to D;
15         record all field access information in D;
16         for static field access add extra information to D;
17     end
18     insert the information in E into DB_E;
19     insert the information in I into DB_I;
20     insert the information in D into DB_D;
21 end
```

Algorithm 7.1: The High Level Algorithm for Generating Access Dependency Graph

- `dependencies` – a list for storing all the dependency information
- `pairCount` – an integer variable for storing the number of entities
- `classPath` – an array of Strings representing the classpath
- `unjarPathPrefix` – the path (String) where jar and zip files are extracted to class files
- `classOutputPrefix` – the path (String) where XMLs generated from class files are kept and some other constants, some of which holds some extra information about entities (e.g. static or non-static) while some others holds names for database stored procedures or functions. The detailed meaning of all these will be apparent as we discuss our full process.

7.7.4 Generating XML Files

For generating XML files from class files, we use the *ClassReader* tool just as the same way we used in Chapter 6. We have a class `FileManager` whose method `parseClassPath` iterates through all the directories in the classpath does two things: (1) If it finds any class file, it generates XML file using the `XMLGenerator` class (which internally uses *ClassReader*) and puts the XML file inside a subdirectory in directory `classOutputPrefix`. The subdirectory is named according to the directory where the actual class file resides. (2) If it encounters any jar or zip files, it extracts them into directory `unjarPathPrefix` (and creating appropriate subdirectories inside) using its other private methods `lookForJarAndZips` and `injar`.

7.7.5 Building up the Entity and Inheritance Information

Just to alleviate all confusion, by an entity we mean either a method or a field (though for the sake of building the dependency relation later on, we also store class names in our entity list). Technically it is the class name followed by a colon followed by a method signature, or the class name followed by a colon

(:) followed by a field name (in case of only class name entries, there is only the class name, no colon). The list of entities in necessary before we can start building the dependency relation (or graph) because the accessor-accesse pairs (edges) of the relation (graph) are based on nodes from the entity list.

We have a class `EntityHandler` that builds up the entity list. The `buildEntites` method of this class iterates through all the generated XML files one by one and parses them to fetch out class name, superclass name, interfaces, methods and fields. Before parsing the methods and fields, it passes the superclass and interfaces information to the class `InheritanceResolver`. For keeping track of the inheritance information we maintain a hashmap data structure `inTable` which maps a class name to an instance of class `Vertex`. Class `Vertex` has the following members:

```
String className
Vertex superClass
Edge subList
Set<String> transitiveChildren
```

`className` represents the name of the class. `superClass` is the superclass of the class and `subList` begins a linkedlist representing all the direct subclasses or implementations of the class (or interface). `sublist` is actually an instance of class `edge` which has the following members:

```
Vertex vertex
Edge nextEdge
```

The `Edge` class sets up the link between a class(or interface) and its direct subclasses (or implementations). The subclasses themselves are linked by further instances of `Edge` represented by the `nextEdge` member. So effectively we have a graph structure representing the inheritance (and implementation) relationship among classes. We build it this way so that we can calculate the transitive subclasses and implementations of a class or interface easily, which is represented by the `transitiveChildren` member. This `transitiveChildren`

member is needed later for adding extra edges for the dynamic binding problem. For example, if class A has subclasses B, C and D and class B has subclasses E and F, then class A will have B, C and D in its sublist, separated by the `Edge` links, and B will have E and F in its sublist, similarly separated by the `Edge` links. Following the `subList` links recursively, we can calculate A's descendants B, C, D, E and F and so on. Figure 7.6 demonstrates the graph structure of the inheritance hierarchy.

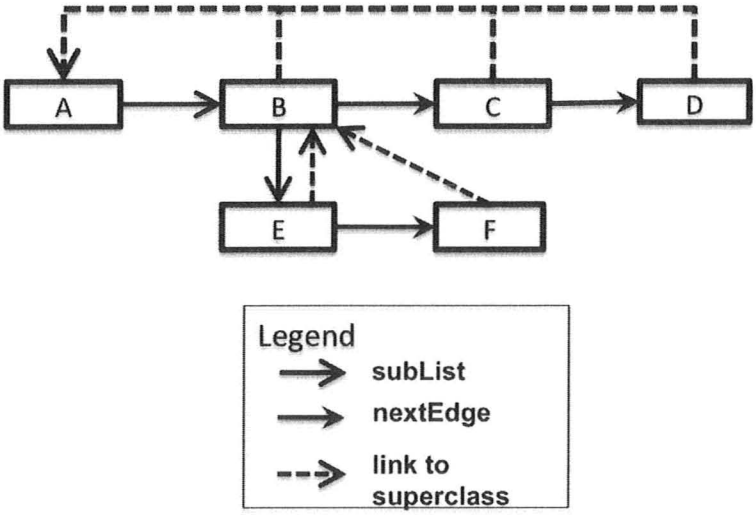


Figure 7.6: Graph Structure of the Inheritance Relation

We got 4 methods inside the `InheritanceResolver` class. Method `checkInheritance` checks for superclasses and interfaces and invokes method `makeInheritanceEdge`, if the superclass is not empty and the superclass is not `java.lang.Object`. We don't consider `java.lang.Object` in our inheritance hierarchy because every class in Java is implicitly a subclass of it. The `makeInheritanceEdge` method adds necessary edges to the inheritance graph. The other two methods are `calculateTransitiveChildren` and `getTransitiveChildren` which are used for calculating descendants by recursively following the graph links described above. The following code segment from `getTransitiveChildren` demonstrates this:

```
...
Edge edge = vertex.subList;
```

```
        while(edge != null){
                list.add(edge.vertex.className);
                if(edge.vertex.transitiveChildren == null){
                    list.addAll(getTransitiveChildren(edge.vertex));
                }
                else{
                    list.addAll(edge.vertex.transitiveChildren);
                }
            edge = edge.nextEdge;
        }

        vertex.transitiveChildren = list;
...
```

Coming back to our **EntityHandler** class, we have methods **parseClassForMethods** and **parseClassForFields** that parses the XML for methods and fields, respectively. Every method and field we encounter is put in a hashmap called **entities**. This hashmap has the entity name (class name : method signature / field name) as key and an integer as value. We maintain this integer in a shared integer variable called **entityCount**. This variable starts from 0 and whenever we find an entity we increment it by one and store it in the **entities** hashmap against the entity name key. For example, for the following two example classes, the **entities** hashmap will look like Table 7.7.1.

```
class A{
        public void a(int i){

        }

        public int test(){
                return 0;
        }
}

class B{
```

```
        int count = 0;
        public void b(int i){

        }

        public int test(){
                return 0;
        }
}
```

Key	Value
A:A()	1
A:a(int)	2
A:test()	3
B:B()	4
B:b(int)	5
B:test()	6
B:count	7

Table 7.7.1: Sample Entity List [RST+04]

Note that the constructors method for each class appears because although they were not explicit in the source code, they are explicit in the bytecode and hence, in the XML.

In addition, we maintain another hashmap called `entityInfo` that stores information about entities, specially whether an entity is static or non-static. This hashmap has the entity id as key and a flag (indicating staic or non-static etc.) as value. This flag value can have the following possible values stored in our common `Data` class:

`CLASS` (0) if the entity is a class
`NON_STATIC_METHOD` (1) if the entity is a non-static method
`STATIC_METHOD` (2) if the entity is a static method
`NON_STATIC_FIELD` (3) if the entity is a non-static field
`STATIC_FIELD` (4) if the entity is a static field

Although this hashmap is not useful for the dependency graph generation, the information stored in this hashmap might come handy when we have to

deal with maintaining the dependency graph with successive patches, as will be discussed in Chapter 8.

At the end of building the entity list, we calculate the descendants of each class using the `calculateTransitiveChildren` method. This information will be used in adding extra information for handling dynamic binding issue during the dependency relation generation process.

The reader might wonder why we are using a hashmap and storing interger values against them rather than storing the entity names directly in a list. This was rather an empirical software engineering decision. The reasons will be clear when we discuss the dependency relation generation process next.

7.7.6 Building the Access Dependency Relation

Having built the list of entities and gathering the inheritance information, we now begin to build up the dependency relation. Before we move into the details, we first discuss how we intend to store the dependency relation in memory.

7.7.6.1 Storing the Dependency Relation in Memory

In the previous section, we showed how we store the entity information in memory using a hashmap rather than a simple list. The way we store the dependency relation in memory is complementary to that. Its worth mentioning that our first approach was to store entity information in a list with String values as members of the list. Theoretically there is no problem with that. But if we store the entity information in a list with String values, that means we have to keep the full names of entities (class : method/field) in the dependency relation as well. For example, consider the following code segment:

```
class A{
        public void test(){
                B b = new B();
                b.doNothing();
        }
}
```

```
class B{
        public void doNothing(){
        }
}
```

For this two classes, if we had stored the entities in a list and built the dependency relation accordingly, they would look like the ones shown in Table 7.7.2.

Entity
A:A()
A:test()
B:B()
B:doNothing()

(a) Entity List

Accessor	**Acessee**
A:test()	B.B()
A:test()	B.doNothing()

(b) Dependency Relation

Table 7.7.2: Plain Entity List and Corresponding Dependency Relation

For small size programs, this would work quite fine. But remember we are dealing with a system having hundreds of thousands of class files and possible millions of enitites; with the fact that everything we are doing has to be done keeping impact analysis in mind. When first built our entity list and dependency relation like Table 7.7.2, and inserting those into the database, it was taking up a huge memory and also a huge table space in the database. Remember by class name we mean the fully qualified class name and for methods, we take the full method signature (not only the name), some of the class names turned out be almost 500 characters long and some of the method signatures turned out to be over 2000 characters long. And since there can be multiple method calls or multiple field access from the same method, in the dependency relation, there would be repetitions of such long names again and again. This kind of information storage also makes the impact analysis (which is being done by my colleague Wen Chen [4]) harder. During impact analysis, when we tried to fetch the information from the database and run analysis on

[4](in progress)

memory, these huge and repeatedly occurring string values made the impact analysis process horribly slow.

That is why we finally decided to store the entity information and dependency relation in a different way. That is, we decided to assign unique integer id's to each entity and in the dependency relation, we only keep the id's in pairs. For the example shown above, we get storage like Table 7.7.3.

Entity	Id
A:A()	1
A:test()	2
B:B()	3
B:doNothing()	4

(a) Entity List as Hashmap

Accessor	Acessee
2	3
2	4

(b) Dependency Relation with Id's

Table 7.7.3: Entity List (as Hashmap) and Corresponding Dependency Relation

Storing this way dramatically reduces the memory requirement because we are now using integer Id's instead of long strings repeatedly. An added advantage of this hashmap is that during the dependency relation generation, if an entity is found which was already found before, we can just look up the hashmap with that entity's string value to retrieve its id, which takes very negligible constant time and boosts the performance improvement. This also makes the impact analysis phase extremely faster because the database tables (where we finally store all these information, discussed later) also have the same format of storage – entity and id against them. Assigning id's also opens an even more elegant way of storing the dependency relation pairs in memory. In Java, an integer is 32 bits, whose maximum value is more than 2000 million, which is a reasonably safe upper limit. But for storing the accessor and accessee id's as relation pairs in memory, we can use a java long variable which is 64-bits, storing the accessor id in the upper 32-bits and the accessee id in the lower 32-bits as shown in Figure 7.7.

Thus, for storing the dependency relation, we can just keep a list of longs. In our program we call this list `dependencies`. Just to demostrate, below is the code segment we use for doing it:

...

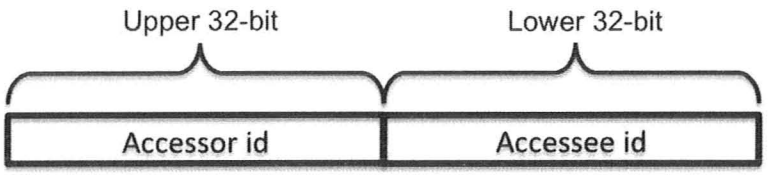


Figure 7.7: Storing id's in Java long variable

```
long dependency = c1;
dependency = dependency << 32;
dependency += c2;
dependencies.add(dependency);
...
```

Now we move into the details of the process.

7.7.6.2 The Process of Building Dependency Relation

Our `Dependencyhandler` class has a method `buildDepepndecyRelation` that iterates through all the XML files and calls another method `parseMethods` which parses the methods of a particular class for finding dependency instructions, i.e., calling a method or a field access. Since calling a method can be done in four ways (*invokeinterface, invokespecial, invokestatic, invokevirtual*) and accessing a field can be done in four ways (*getstatic, putstatic, getfield, putfield*), it takes the instructions with any of these eight kinds of accesses and records the accessor entity and the accessee entity. Just to clarify once again, we store the accessor entity as *Fully Qualified Accessor Class Name : Signature of the Accessor Method* and the accessee entity as *Fully Qualified Accessee Class Name : Signature of the Accessee Method / Name of the Accessee Field.*

Since, our entity list is a hashmap, and our dependency relation is pairs of id's, whenever we add a new pair to the relation, we just extract the id's of the accessor and accessee entity using the `get` method of the hashmap, which makes it pretty fast. Notice that this is another advantage of using hashmaps.

Since the same method can be called or same field can be accessed multiple times from a particular method, to avoid duplicacy, we maintain a temporary set of accessed entities during the parsing of each method. If any accessed

method or field is already in the set, we ignore it. Thus we get rid of duplicate relation pairs occurring in our dependency relation.

Now, there are a number of side points here. Remember we need to know record additional information in case of method calls that are dynamically bound. And we mentioned previously that *invokevirtual* and *invokeinterface* are the candidates for this. So we record this information with the following code snippet:

```
...
if(instruction.startsWith("invokevirtual") ||
instruction.startsWith("invokeinterface")){
        virtualOrInterface = true;
}
...
```

Besides, remember when we add extra dependency pairs to the relation, we need to know which transitive subclasses or of a certain class overrode a called method, and which transitive implementations of an interface have their own version of a called method. Finding this is not so hard. Because when we parse call or access instructions, if we find a called method that is not in our entity list but the class, to which that method belongs, is in the entity list, then that class uses the inherited version of the method and has not overriden that method itself. We maintain another list (actually another hashmap) called `nonOverridenEntities` to keep account of these entities. It is better demonstrated with an example. Consider the following code segment:

```
class A{
        public void one(){
                ...
        }

        public void two(){
                ...
        }
}
```

```
class B extends A{
        public void one(){
                ...
        }
}

class C{
        public void test(){
                B b = new B();
                b.two();
        }
}
```

Here, class B extends class A but overrides only method *one*(), not *two*(). So B : *two*() won't be in our main entity list. But since class C's method *test*() calls class B's method *two*(), we put B : *two*() in our `nonOverridenEntites` list.

Later on, if we find a dependency and if that dependency is a candidate for the dynamic binding problem, we use the descendants, superclass information and the `nonOverridenEntites` to add the extra dependency pairs. Instead of keeping these extra dependency pairs in the same ArrayList `dependencies`, we maintain another ArrayList `extraDependencies` of long values. This is done with the help of two methods `addImplicitAccessDependencies`, which adds dependency edges (pairs) to the subclasses and implementations which have defined their own version of inherited methods, and `addEdgeToClosestSuperClass`, which adds a dependency edge to the method of a closest transitive superclass where the method body is defined in case the called class has not defined the method itself. We also maintain another list `dynamicClients` containing the id's of the methods whose dynamic binding issue have been resolved, so that they are not considered again. It is worth mentioning that, for only building the dependency relation once, we did not actually need to different ArrayList in memory. We could just keep all dependency pairs in one single list. However, in Chapter 8, when we discuss how

to maintain the dependency relation with successive patches, the reason for keeping them separate will become apparent.

Again, we need to know whether an access is to a static field or not, because in case of a static access we have to explicitly include a dependency pair (edge) to the *<clinit>* method of that class. So we record for static accesses as well:

```
...
if(instruction.startsWith("getstatic")||instruction.startsWith("putstatic")){
          ...
      isStaticField = true;
}
```

And later on we add the dependency on the *<clinit>* method using `addDependencyToStaticInitializer` method.

Thus we have all the dependency information in our list of Java long values, where each value is an encoded dependency pair (edge).

7.7.7 Saving the Information Into Database

Having built up all the information in memory, our next task is to store everything in database. Before storing we need to prepare our database.

7.7.7.1 Preparing the Database

We are using Oracle 11g database in our project. Although any other standard database (e.g. SQL Server, MySQL) would have been suitable, we are using it because we have easy access to it in our university's oracle server.

We need different tables for storing different kinds of information. For storing the entities, we have a table `Entity` with three columns – `Class_Name`, `Feature_Name` and `Id`; the first of them are of data type varchar and the last one . `Class_Name` is the name of the class and `Feature_Name` is either the signature of the method or name of the field. `Id` is the unique id assigned to each entity. This table is populated from our in memory hashmaps `entities` and `nonOverridenEntities`.

For storing the dependency relation, we have three tables called `Dependency_Relation_Original`, `Dependency_Relation_Extra` and `Dependency_Relation`, each consisting of two integer columns – `Accessor_Id`

and `Accessee_Id`. This first table is populated from our in memory ArrayList `dependencies`, while the second one is populated from our in memory ArrayList `extraDependencies`. The third table is a union of the first two holding all dependencies. To maintain referential integrity, we keep foreign key constraints from `Accessor_Id` and `Accessee_Id` to the `Id` column of `Entity` table.

We also store the meta-data about entities,i.e, the information from the `entityInfo` hashmap, into a database table called `Entity_Info`, which has two columns – `Id` and `type`. `Id` is the id of the entity and `type` is the type of the entity corresponding to the value field of the hashmap `entityInfo`.

Finally, we also store the inheritance information into two database tables. We have a table `Inheritance` with two varchar columns – `parent` and `child`. This table stores the direct subclass or implementation information of a class or interface. This corresponds to the `subList` member of our `Vertex` class. We have another table `Inheritance_SuperClass` with the same schema structure but this one only stores superclass information.

Apart from the tables, we also have a number of oracle stored procedures for inserting the information into the tables:

- `SP_INSERT_ENTITY_ORIGINAL` – for inserting entities into the entity tables `Entity_Original`
- `SP_INSERT_DEPENDENCY_RELATION_ORIGINAL` – for inserting dependency relation pairs (edges) into the `Depepndency_Relation_Original` table
- `SP_INSERT_DEPENDENCY_RELATION_EXTRA` – for inserting extra dependency relation pairs (edges) into the `Depepndency_Relation_Extra` table
- `SP_COMBINE_DEPENDENCY_RELATION` – for combining dependency relation pairs (edges) from tables `Depepndency_Relation_Original` and `Depepndency_Relation_Extra` into the `Depepndency_Relation` table
- `SP_INSERT_ENTITY_INFO` – for inserting meta data about entities into `Entity_Info` table

- SP_INSERT_INHERITANCE – for inserting direct subclass and implementation information into Inheritance table
- SP_INSERT_INHERITANCE_SUPERCLASS – for inserting superclass information into the Inheritance_SuperClass table

7.7.7.2 Inserting the Data into The Tables

We have two classes DatabaseHelper and ConnectionManager to help us deal with the database. ConnectionManager is the class for getting us database connection and closing it. DatabaseHelper has four methods to help us inserting different kinds of data into the database.

First, insertEntities method of class DatabaseHelper is called to store our in-memory original entity lists entities and nonOverridenEntities into database table Entity_Original using SP_INSERT_ENTITY. The insertEntityInfo method is then used to populate the meta data about entities in the Entity_Info table using SP_INSERT_ENTITY_INFO.

Next, insertDependencyRealtion method is called to store the in-memory dependency information dependencies into the database table Depepndency_Relation_Original using SP_INSERT_DEPENDENCY_RELATION. The same method is also used to store the in-memory dependency information extraDependencies into the database table Dependency_Relation_Extra using SP_INSERT_DEPENDENCY_RELATION_EXTRA. Then the insertDependencyRealtion method is used to combine the relation pairs from tables Depepndency_Relation_Original and Dependency_Relation_Extra into table Dependency_Relation with the help of SP_COMBINE_DEPENDENCY_RELATION.

The insertInheritance and insertInheritanceSuperclass methods are used to insert inheritance information into the database tables using SP_INSERT_INHERITANCE and SP_INSERT_INHERITANCE_SUPERCLASS, respectively.

Thus we have all the in-memory information stored in our database tables. The information stored in the Entity and Dependency_Relation would be used in the impact analysis phase. Although impact analysis and test selection is not one of the main subject matters of this thesis, in the following

section we discuss a bit how the impact analysis will work based on our stored information.

7.7.8 Performance and Statictics

We ran our process on our university's oracle server machine which has 8 quad core 3,2 GHz processors and 32 GB memory. The operating system is Linux 64-bit. On that machine, the XML generation for all the class files, including the ones from jars and zips (total 230,000) takes approximately 40 minutes. The computation in memory, namely generation of the entity list, inheritance information and dependency relation takes just more than one and a half hours. Finally, the insertion of all those information into database takes approximately four and a half hours. So in total, the whole process takes approximately 7 hours dominated by the database insertion time, which is reasonably acceptable for any customer who spends weeks for running all their tests.

There are almost 4.5 million methods and fields (nodes) in total, and the dependency relation (graph) has almost 12 million accessor-accessee pairs (edges). These numbers give an idea on how huge the graph is.

7.8 Impact Analysis

We can consider the `Entity` table as a set of nodes and `Dependency_Relation` table as the set of edges of the access dependency graph. Along with these data, the modification information between versions of class files (from Chapter 6) are also input for the impact analysis phase. The impact analysis starts from a function (method) or a field that has changed, extract its id from the `Entity` table and then traces back along its accessors using the `Dependency_Relation` table. Thus it builds up an impact set, which is the transitive accessor set of that changed entity. Consider the original and modified versions of a program in listings Listing 7.7 and Listing 7.8.

```
class A{
        private void m1(){
                B b = new B();
```

```
                b.test();
        }

        public void m2(){
                m1();
        }

        public void m3(){
                ...             // doesn't directly or
                    transitively access B:test()
        }
}

class B{
        private int i = 0;

        public void test(){
                i++;
        }
}
```

Listing 7.7: Original Version of Code

```
class A{
        private void m1(){
                B b = new B();
                b.test();
        }

        public void m2(){
                m1();
        }

        public void m3(){
                ...             // doesn't directly or
                    transitively access B:test()
        }
}

class B{
        private int i = 0;
```

```
        public void test(){
                i--;
        }
}
```

Listing 7.8: Modified Version of Code

The `Entity` and `Dependency_Relation` generated for this program will look like this:

Class Name	Feature Name	Id
A	A()	1
A	m1()	2
A	m2()	3
A	m3()	4
B	B()	5
B	test()	6
B	i	7

(a) Entity Table

Accessor Id	Acessee Id
3	2
2	5
2	6

(b) Dependency Relation Table

Table 7.8.1: Entity Table and Dependency Relation Table

Here, only the B : $test()$ method is modified and its id is 6. So using the `Dependency_Relation` table, the transitive accesor id's 2 and 3 are found which correspond to A : $m1()$ and A : $m2()$, respectively. So these are found in the impact set.

7.9 Test Selection

Remember we are interested in finding the functions in the customer's application layer which are potentially affected, i.e., the functions in the customer's application layer that directly or transitively access the changed functions or fields. Based on this information, the required tests would be selected from the existing test suite. Although at this moment, we do not have too much idea about the hypothetical customer's test suite, we are presuming they have a coverage test matrix telling which functions are covered by which tests. For

example, for the example in Listing 7.7, we might have a coverage matrix like Table 7.9.1, assuming class A is a class from the customer's application layer.

Test Name	Class Name	Function Name
Test1	A	m2()
Test2	A	m3()

Table 7.9.1: Test Coverage Matrix

Here we have $Test1$ covering method $A : m2()$ and $Test2$ covering method $A : m3()$. Taking the intersection of this information with the impact set we calculated, only $Test1$ is selected.

7.10 A Critical Impact Analysis Issue

We have discussed the details of the access dependency graph generation process and also demonstrated in short how impact analysis and test selection can be carried out on this later on. But some interesting questions are left. After applying a patch, we have a modified dependency relation. So which version of the dependency graph do we carry out our impact analysis on? The old one or the new one ? Or do we need to compare the two versions of the graph ? In this section, we discuss these issues.

A change inside a method made by a patch, by altering or adding or deleting some non-access instructions[5] inside it, does not alter our dependency graph. So in these, it does not really matter which version of the graph we actually use. The situation is more interesting when the change occurs in access instructions. Consider Figure 7.8. Here, method a previously used to call method b, but after patching it now calls method c instead. So the graph structure has now changed.

Method b might well have been deleted, or might not. But a careful thinking reveals that, in this case, method a will be reported as a changed method by our modification finder tool anyway (Chapter 6). So all we need to do is to begin impact analysis from method a. It does not really matter how the graph structure changed. So we can just work with the new version of the

[5]non-access means instruction other than method calls or field access

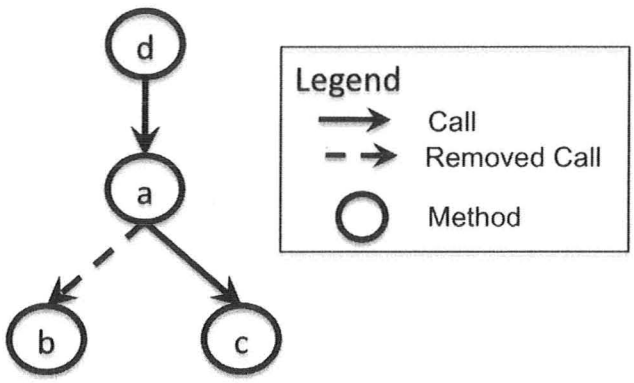


Figure 7.8: Call relation among methods

graph here. Similar reasoning applies for a newly added method as well. If a method is newly added, then there must have been a change in some other method who is now calling that newly added method. And we will have that other method reported as a changed method by our modification finder tool. Again, the new version of the graph is sufficient for this.

The only interesting issue left is if an overriden method is added without changing the call structure elsewhere. Consider the following code segment:

```
class A{
        public void m(){
                ...
        }
}

class B{
        public void m(){     // overriden version of m()
                ...                          //  newly added
        }                              //  by a patch
}

class C{
        public void test(){
                B b = new B();
```

```
                b.m();
        }
}
```

Here the the overriden version of method $m()$ has newly been added by the patch. Before applying the patch, the dependency graph will have the following edges:

$C : test() \rightarrow B : m()$
$B : m() \rightarrow A : m()$

The second edge is due to the implicit dependency to the nearest transitive superclass version of the method.

After applying the patch, the dependency graph will have only the first edge. We no more have the edge to the nearest transitive superclass's method because $m()$ is not defined in B. The important thing is that impact analysis is still possible beginning with $B : m()$, since it is a newly added method.

Considering everything, we have come to the conclusion that only the version of the dependency graph after applying the patch is enough for impact analysis.

7.11 Other Impact Analysis Issues

In this section, we discuss some other issues pointed out in impact analysis pointed out in related works, especially in [RST+04]. Notice that with the modification finding tool from Chapter 6 and with the dependency graph generation tool discussed in this chapter, it is possible to carry out impact analysis. As we discussed section 7.1, atomic changes like changes in initializers, constructors, fields are all reported as change in certain methods by our modification finding tool. And then we can use our dependency graph for the impact analysis. Some other issues pointed out in [RST+04] and how we handle them are discussed in the following subsections.

7.11.1 Changes in Inheritance Hierarchy and Overloading

It is possible for changes to the inheritance hierarchy to affect the behaviour of a method, whose code is not changed. Consider Figure 7.9 taken from [RST+04]. Various constructs in Java such as `instanceof`, casts and exception `catch` blocks test the run-time type of an object. If such a construct is used within a method and the type lies in a different position in the hierarchy of the program before the edit and after the edit, then the behaviour of that method may be affected by this hierarchy change (or restructuring). For example, in Figure 7.9(a), method $foo()$ contains a cast to type B. This cast will succeed if the type of the object pointed to by a when execution reaches this statement is B or C in the original program. In contrast, if we make the hierarchy change shown in Figure 7.9(b), then this cast will fail if the run-time type of the object which reaches this statement is C. Note that the code in method $foo()$ has not changed due to the edit, but the behaviour of $foo()$ has been possibly altered.

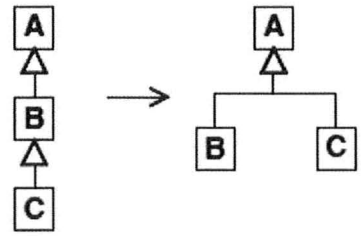
In case of a change in inheritance hierarchy change, like the change in a superclass or interface, our modification finding tool will report it as a superclass or inheritance change. For example, in the example just mentioned, class C will be reported as a changed class with superclass changed. And our dependency graph built using the new version of class C will have that changed incorporated in itself. Then our impact analysis can just begin with any method (including $<init>$) of C that has been referred from somewhere to find the potentially affected methods.

7.11.2 Threads and Concurrency

Another issue pointed out by [RST+04] is threads and concurrency. But note that just like [RST+04], changes related to threads like addition/deletion of `synchronized` blocks and the addition/deletion of `synchronized` modifiers on methods will both result in a changed method being reported by our modification finding tool, because this is a change in the 16-bit access flag. Then our impact analysis can just begin with those methods using the dependency graph.

```
class Test {
  public void foo( ){
    A a = new C();
    ...(B)a...
  }
}
```

(a)



(b)

Figure 7.9: Hierarchy changes that affects a method whose code is not changed [RST+04]

7.11.3 Exception Handling

Just like [RST+04], exception handling is not a significant issue in our analysis. Any addition or deletion or statement-level changes to a `try`, `catch` or `finally` block will be reported as a changed method. Then our impact analysis can be carried out using that method and the dependency graph.

7.11.4 Changes to CM and LC

Remember from section 7.1 that **CM** is a changed method and **LC** is a change in virtual method lookup as discussed there. Some additional issues pointed out by [RST+04] are (i) adding a body to a previously abstract method, (ii) removing the body of a non-abstract method and making it abstract, or (iii) making any number of statement-level changes inside a method body or any method declaration changes (e.g., changing the access modifier from public to private, adding a synchronized keyword or changing a throws clause). Notice

that in all these cases, the corresponding method would be reported as a changed method in our analysis.

In addition, in some cases, changing a method's access modifier results in changes to the dynamic dispatch in the program (i.e., **LC** changes). For example, there is no entry for private or static methods in the dynamic dispatch map (because they are not dynamically dispatched), but if a private method is changed into a public method, then an entry will be added, generating an **LC** change that is dependent on the access control change, which is represented as a **CM**. However, in our case, this kind of change will still cause the method to be reported as a changed method and using the newer dependency graph and the change information, impact analysis can still be successfully carried out.

7.12 Summary

In this chapter, we discussed the concepts and building process of the access dependency graph generation process. We also discussed how the modification information from Chapter 6 and the dependency graph from this chapter aids tacking several impact analysis issues. With this chapter, the discussion of the main subject matters of this thesis comes to an end.

However, in the next chapter, we discuss some of our ideas and thoughts about the maintenance of the dependency graph in the future, with the application of successive patches.

Chapter 8

Maintaining the Dependency Graph Over Time

In this chapter, we discuss some ideas and issues about maintaining our dependency graph over time with the application of patches. The topics discussed in this chapter are not the main subject matter of this thesis. However, it is worth discussing these issues, since the future maintenance of the dependency graph is of obvious interest. All the ideas discussed in this chapter are at purely hypothetical level.

8.1 The Maintenance Issue

We first discuss what we actually mean by maintaining the graph. Say we have a certain version of the dependency graph. When a patch is applied, it changes some of the files, adds new files and may even delete some old files. So our dependency graph structure is very likely to change. And we have to make sure that after patching, we have the correct consistent version of the new graph. There are two possible ways we can do that:

- Build the whole dependency graph again after patching
- Amend the existing dependency graph based on the modification information

At first thought, amending the existing dependency relation seems easy enough, and also seems a logical way to go, rather than following the time consuming way of building the whole call graph again. Theoretically that makes more sense too. However, we have given it a considerable amount of thought and reached the fact that in the worst case, amending is probably no better than building the whole graph again. We discuss our thoughts one by one to demonstrate how we reached that conclusion.

8.2 Patch Complexity

As discussed in Chapter 2, we examined several of the Oracle patches and the highest number of files in a patch was found to be as big as 6,000 (and may be more). Now the question is, how complex can a modification done by a patch be? For example, a patch can be as simple as just modifying some instructions inside one or a few functions (methods), or it can be as complex as modifying lots of methods, adding or deleting existing methods, introduce changes in the inheritance hierarchy of classes etc.

Since we are concerned about building automated tools, we have to take into account the worst case complexity of patches, that is, we have to assume every kind of possible changes that can be made by a patch.

Besides, remember since we store our dependency graph information in the database, maintaining it will involve considerable interaction with the database. The main problem with maintenance is introduced by the following things:

- Changes in virtual or interface method call
- Changes in the inheritance hierarchy
- Changes in type of method call or type of field access

We now discuss these one by one.

8.3 Changes in Virtual or Interface Method Call

Recall that virtual and interface method calla are carried out using the *invoke-virtual* and *invokeinterface* instructions. Consider an example that demonstrates the problem. Consider Figure 8.1 where class B and class C has separate inheritance hierarchies.

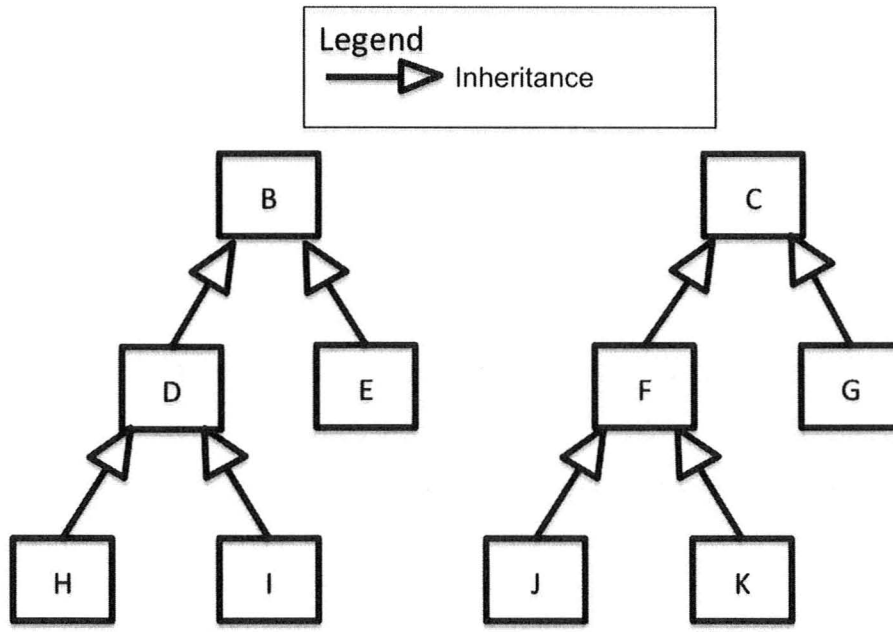


Figure 8.1: Inheritance Hierarchies

Now consider there is another class A and method $A : test()$ calls method $B : m()$. Then for our access dependency graph, besides this original call edge, we also have edges from $B : m()$ to the $m()$ method of those subclasses of B that overrode $m()$. Suppose now that the patch makes a change inside $A : test()$ such that it now calls method $C : m()$ instead of $B : m()$. In this case, to maintain the graph, we need to delete the edge $A : test() \rightarrow B : m()$ but we cannot just delete the edges from $B : m()$ to its subclasses. Because there might be other methods in other classes in the system that still hold calls to $B : m()$ and for them, those edges are still needed.

We have two choices here for– as for the first option, we can search through the dependency graph whether any other method holds the call references to $B : m()$ and if not, delete those extra edges as well. As you can guess, this option will be extremely costly because of the searching time. For the second option, remember we have two separate tables in our database `Dependency_Relation_Original` and `Dependency_Relation_Extra`, the first one holding the original call edges and the send holding the extra edges added for handling dynamic binding. So we can just delete edges like $A : test() \rightarrow B : m()$ from `Call_Relation_Original` and once we are done with all of these kinds of edges, we can search `Dependency_Relation_Extra` to see whether any `Accessor_Id` exists for $B : m()$. If not, we can safely delete all those extra edges from `Dependency_Relation_Extra` as well. Otherwise, we just keep them. This option is much less costlier than the first one. Notice, however, that both options involve lots of database interaction.

8.3.1 Changes to the Inheritance Hierarchy

Changes to the inheritance hierarchy can be introduced in two ways – (1) Changing the superclass of an existing class and (2) Introducing a new class whose superclass is an existing class. Actually, the situation can be even more complex when we consider the transitive superclasses and subclasses.

Consider Figure 8.2. Here classes B and C have been removed from the inheritance hierarchy and classes E and F have been introduced. Notice that, nothing inside class A's code might have changed, despite the change in the inheritance hierarchy. So A might not have been reported as a changed class, nor as an added class. Nevertheless, virtual and interface calls to A's methods still need to be re-organized. For example, if there is another class X and $X : test()$ calls $A : m()$, then besides the edge $X : test() \rightarrow A : m()$, we previously had edges $A : m() \rightarrow B : m()$, $A : m() \rightarrow C : m()$ and $A : m() \rightarrow D : m()$, if B and C had overriden $m()$. But now, we have to delete the first edges and instead have edges $A : m() \rightarrow E : m()$ and $A : m() \rightarrow F : m()$. To achieve these, we have to utilize our database tables `Dependency_Relation_Original`, `Dependency_Relation_Extra` and `Inheritance`. Remember the last table stores subclass and implementation information.

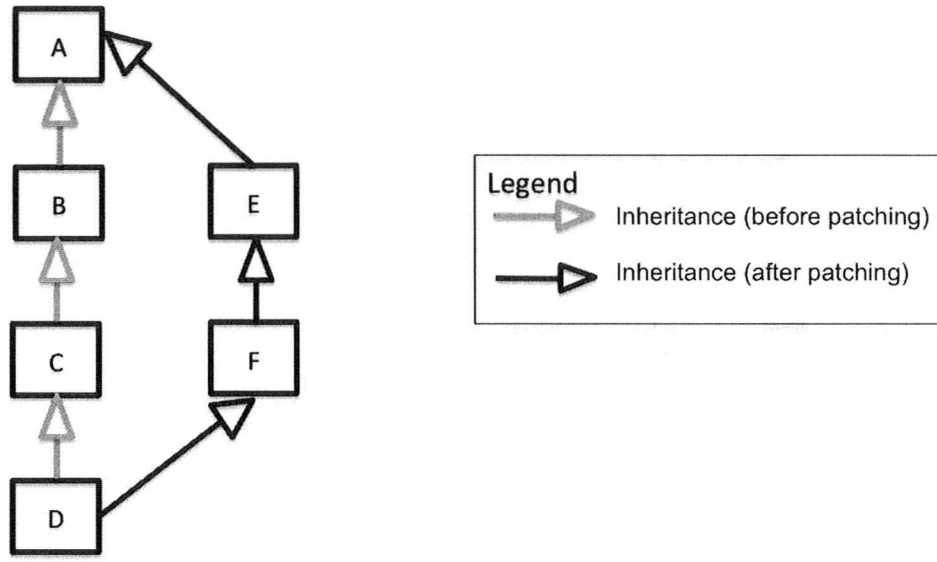


Figure 8.2: Inheritance Hierarchy Change

Similar arguments apply in case of the edges to the nearest transitive superclass. Consider Figure 8.3 where class B didn't override method $m()$ before patching, but after patching it does. So a virtual method call to $C : m()$, assuming C doesn't override $m()$, previously resulted in an extra edge $C : m() \rightarrow A : m()$, but after patching it should result in the extra edge $C : m() \rightarrow B : m()$, without having anything inside class C to change. Again, for resolving this, we have to consult database tables `Dependency_Relation_Original`, `Dependency_Relation_Extra` and `Inheritance_Superclass`. Remember the last table stores superclass information.

This implies that, in worst case, not only we need to worry about the changed or added classes, but also unchanged existing classes that may have changed classes and added classes as their new direct or transitive subclass or superclass.

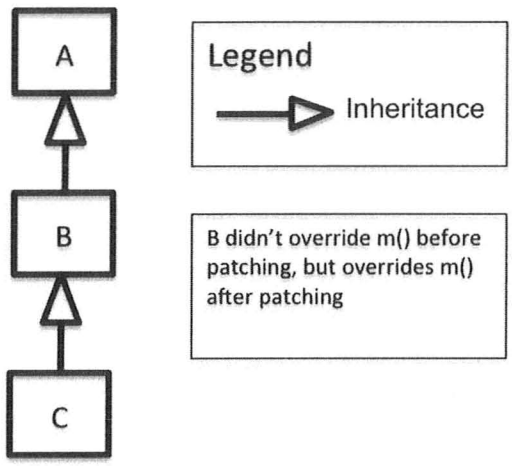


Figure 8.3: Change in Overriding

8.3.2 Changes in Type of Method Call or Type of Field Access

Consider a method $m()$ of a class A which was previously non-static but has changed to static after patching. Say there is another class X and method $X : test()$ calls $A : m()$. Before patching, this call was virtual. But after patching this call is now static. So, although the edge $X : test() \rightarrow A : m()$ remains in the new version, the extra edges from $A : m()$ to subclasses of A now needs to be removed because this call is no longer a candidate for dynamic binding. Similarly, if $m()$ was previously static but now turned non-static, we need to add those edges. Once again, we need the dependency and inheritance related tables from database to resolve this issue.

Again, consider a static field changed to non-static. Previously for an access to that field, we would have an edge to the $<clinit>$ method of the class which that field belongs to. But now since the static reference is no more there, we may need to eliminate the edge to the $<clinit>$ method, unless of course, there are other static field references still out there to that class. So we have to check whether any static references are still left, and if not, then remove all the corresponding reference to the $<clinit>$ method. We can do this with the help of our database table `Entity_Info` which stores meta data

about entities (static, non-static).

There are other issues in the graph amending process as well. Before we move onto those, we examine the abstract steps involved in the amending process.

8.4 The Abstract Amending Process

The steps in the amening process in short are as follows:

1. Remove all deleted classes and their methods and fields from the entity list
2. Remove all deleted methods and fields from changed files from the entity list
3. Adjust the inheritance record according to the removed classes
4. Adjust the dependency relation according to the removed methods and fields
5. Add the classes, methods and fields from the added classes to the entity list
6. Add the classes, methods and fields from the changed classes to the entity list
7. Keep records of change in inheritance hierarchy during the previous two steps
8. Adjust the inheritance record according to those added classes other change in the inheritance hierarchy
9. Adjust the dependency relation according to those added classes, methods and fields
10. Adjust the dependency relation according to the added calls, removed calls, added access and removed access inside changed class files

8.5 Other Issues

Besides the ones mentioned in the section 8.2, there are some other side issues in maintaining the graph. These issues are actually complementary to the ones described there.

8.5.1 Keeping the Data

Remember that our dependency tables only contain id's, the original names (class name, method name, field name) of the entities are contained in the entity tables. So while making amendments to the dependency tables, like the amendments discussed in the previous section, we also have to make changes to the entity tables. For example, if a method is deleted by a patch, we have to first remove it from the entity table and then remove the associated dependencies from and to that method from the dependency tables. Otherwise the tables will be inconsistent.

Theoretically, it does not seem to be that much of a problem. The concept is that whenever an added or deleted entity is encountered, just make changes in the entity and dependency list and keep them consistent. However, problems arise in our empirical domain. We see a trade-off between two options:

- Updating the tables in database
- Bringing the table data into memory, make the amendments there and then insert back into database when everything done

Consider the steps mentioned in section 8.4. They involve deleting and inserting entities into the entity list, dependency relation list and inheritance list. Now all these can be done inside the database without bringing any data into memory. But in the worst case, that may involve lots of application-database interaction. For example, if we want to know the id of a certain entity (method, field), we have to execute a query with that entity, such as

```
select id from ENTITY where feature_name = some_name
```

Again, for example, when we want to know about the subclasses or implementations of a certain class, we also need to execute similar queries which

may take a lot of time if number of changes is reasonably high. Calculating the transitive children in database is not so straightforward either as it is in an in-memory graph structure.

The other way is to bring in the stored entity list, dependency relation and inheritance list into memory, using build in-memory abstract data objects like hashmaps, arraylists and use them for intermediate processing; and finally insert those back into database after all processing has been done. This way the only significant time we need to worry about is the fetching and insertion time. However, the in-memory processing is much more elegant and faster. For example, we can keep in memory hashmaps for the entity list, arraylist for the dependency relation and a graph structure for the inheritance hierarchy, just as we did in Chapter 7 when building the dependency graph. But remember from Chapter 7 that the database insertion of millions of rows is the main villain in our tool performance. So both trade-off options have their own merits and demerits.

8.5.2 Reusing Id's

Another issue in maintaining the dependency graph is in reusing the id's of the entities that are deleted. Because we can assign those id's to newly introduced entities, instead of wasting them. Although as far as the upper limit of the integer id's is concerned (over 2000 million), reusing is not a big issue we can live with even wasting those deleting id's for hundreds of patches. However, as far as an efficient management tool is concerned, in general, we would like to save those id'd for reuse. We can do this by keeping a database table (for trade-off option 1) or an in memory list (trade-off option 2) and populating with id's whenever an entity is deleted. Then when new entities are added, we can first look up that table/list and assign one from it if all the id's in that table/list are not used up by that time.

8.6 The Possible Detailed Amending Process

Having discussed the issues related with amending the graph, we now list the steps of the possible amending process (the ones we mentioned at a high level

of abstraction in section 8.4) in detail. As for choosing between the trade-off options, we choose option 2 (brginging the data in memory first) below.

- First we need the modification information from Chapter 6 and also information about added and deleted files that we get from patch analysis. For the changed files we have to generate the difference holding XML files. And for the added and deleted files we have to generate the class-equivalent XML files.

- We have to bring in the `Entity`, `Entity_Info`, `Dependecy_Relation_Original`, `Dependecy_Relation_Extra`, `Inheritance`, `Inheritance_Superclass` tables from the database into memory and build the entity hashmap, entity meta data hashmap, dependency relation arraylist, inheritance graph structure and superclass linked list.

- We have to create a list holding deleted id's. Say we call it `deleted_list`. Also we have to record the highest id used in the existing entity list in a variable `maxid`

- We have to remove all deleted classes and their methods and fields from the in memory entity hashmap and from the entity meta data and add those deleted id's into `deleted_list`

- We have to remove all deleted methods and fields from changed files from the in memory entity hashmap and from the entity meta data and add those deleted id's into `deleted_list`

- The deleted classes will have caused some changes in the inheritance hierarchy. So we have to adjust the inheritance graph and superclass linked list in memory according to the removed classes.

- We have to adjust the dependency relation arraylist according to the removed methods and fields.

- From the newly added classes, we have to add the class, method and field information to our entity list and to the entity meta data as well.

While assigning id's to the added entities, we look for deleted id's in `deleted_list` first. If that list is empty, then we begin assigning from `maxid`. Also the newly added classes will perform some enhancement in the inheritance hierarchy. So we have to enhance the inheritance graph and superclass linked list accordingly

- The changed files might also contain added methods and fields. We have to add method and field information to our entity list and to the entity meta data as well. For assigning id's we follow the same strategy as in the previous step. If any of the changed classes has its superclass or interfaces changed, it will cause some modification in the inheritance hierarchy. So we have to enhance the inheritance graph and superclass linked list accordingly

- We have to parse the added classes and added methods in changed classes for finding new calls and field accesses and add those to our dependency relation arraylist. We also have to account for the extra call edges for dynamic binding due to the virtual and interface calls.

- We have to look for the existing unchanged classes who have now got a newly added class or a changed class as its new transitive subclass or superclass, or who have lost a transitive sub or superclass due to deletion. For all these kinds of classes, we have to make necessary amendments in their virtual call relation.

- For changed methods, we have to look for newly added calls and field references and adjust the dependency relation accordingly. Also, if the type of a call or field access changes, we have to amend entries for them as discussed earlier.

- If all the static references to a class are removed, we also have to remove the implicit call edge to that class's $<clinit>$ method.

- Finally, we need to evacuate the database tables and re-insert all these data into them again.

8.7 Our Current Decision

The previous section discussed the possible amending process in detail choosing trade-off option 2. Note that, even with choosing trade-off option 1 the complexity of the process might have been pretty high in worst case. As the reader can see, no matter what trade-off option we choose, the worst case amending process is fairly complex in general. Of course, in reality, it may happen that a patch only performs very minor change in one part of the system, not making any change to the inheritance hierarchy, static access, type of method call etc. But the worst case scenario can be as complex as discussed in the previous sections.

Now as far as our problem domain is concerned, remember from the statistics of Chapter 7 that our dependency graph generation process takes approximately 7 hours, which is a pretty tolerable time span in industry. Considering that our customer patch their system once every 2 or 3 months, 7 hours is not a big deal anyway. So rebuilding the whole graph with every successive patch is not a bad choice at all from industrial point of view. Moreover, with all the complexities involved in the amending process, rebuilding also seems safer. As a side gain, notice also that, rebuilding the graph each time eliminates the need for some of the database tables like `Dependency_Relation_Original`, `Dependency_Relation_Extra` and `Entity_Info`. That way we do not have to insert data into and maintain these tables and our dependency graph generation process will get faster.

Considering everything, for the time being, we have decided to rebuild the dependency graph with every successive patch.

8.8 Summary

In this chapter, we have discussed some ideas and issues about maintaining the dependency graphs with the application of successive patches in the course of time. Since everything presented in this chapter is purely hypothetical, it is well possible that in the future, some of our decisions made in this chapter will be reversed (or again, may be not) based on the continual work conducted on this impact analysis project.

The next chapter, which is the final one of this thesis, discusses the contribution and limitations of the work presented in this thesis, as well as a discussion on the possible future works that can be conducted.

Chapter 9

Discussion and Future Work

In this chapter we discuss the contributions we made in this thesis, limitations faced during the research process and future works that may be done on the work presented in this thesis.

9.1 Contribution

Reverse engineering can be seen as an opposite process of compilation. In compilation, lower level representation of the program is created from a higher level representation. In reverse engineering we follow the other way: from lower level code to higher level specification or information. That's why we we follow the process of analyzing Java byecode files to extract the dependency information from them and also to determine change information between two successive versions of a bytecode file. We see our main contribution as adapting the existing off-the-shelf tools in extracting the required information for impact analysis in our specific problem domain. An important contribution of our approach is that we have been able to build up a worst case static dependency relation despite working at a relatively high level of abstraction, like not moving into statement level control flow or data flow analysis. And as we have discussed in Chapter 8, the dependency relation is built up in such a way so that impact analysis can be carried out successfully without missing anything.

The *Dependency Extractor* tool of *Dependency Finder* [Tes10a] generates

a great deal of information that we need. However, due to the fact that it lacks some useful information (like inheritance), we had to use another tool *ClassReader* of the same toolset and adapted its class equivalent XML output according to our need.

The two complete processes shown in Figure 5.1 and Figure 5.2 are fully automated. Interaction among the Java classes are without any human intervention making the processes more robust and easily verifiable. Moreover, due to the modular nature of the processes, if needed, we can temporarily stop some of the sub-processes to see various intermediate outputs. For example, we can print the inheritance graph to console or to a file on disk to check that it is built up correctly.

The tools developed in this thesis are not merely a proof of concept. Unlike many of the related works discussed in Chapter 4, they just do not just exhibit a concept of larger analysis which may be implemented in a larger scale. Rather they are designed as fully usable (though still domain specific) software products keeping rigorous software development methodologies in mind. The domain where we applied our ideas and design is fully empirical (Oracle E-Business Suite) and probably one of the largest software systems in the world. We keep on trying to incorporate more and more software engineering principles to make these tools even more efficient.

The tools also interact well with third party outside tools (from *Dependency Finder*) and are flexible enough to incorporate new features in them. The codes included in the thesis are easily comprehensible and we hope that any fellow student will be able to understand them and translate them into different architecture within a limited time.

9.2 Limitations

Back in Chapter 2, we mentioned that we have limited, for the sake of this thesis, our scope only to *class* files. However, remember that Oracle patches actually contain files with dozens of other extensions. Although we have done some rough work for handling `SQL` and `PL/SQL` files, currently our tools can only handle Java *class* files.

Moreover, even inside the Java environment, the dependency graph we build is capapble of handling only Java entities at this moment. In reality, there can be database accesses from inside the Java methods like executing database queries, stored procedures and functions etc. And those database functions and procedures can have dependencies among themselves as well. So the complete dependency graph actually extends from the Java environment to the database which we have abstracted out from this thesis.

The dependency graph we build is a kind of raw representation of the dependency relation among the entities of the Java environment. It does not currently incorporates any specific patterns in the dependencies (for example, out of a hundred classes it may turn out that ninety of them depends on a few methods of one single class). In addition, because of the huge size and empirical problems discussed in Chapter 7, we did not do a statement level control flow or data flow analysis. Our analysis is roughly at the method level. This way things get a bit simpler but it abstracts out many of the crucial possibilities. For example, a test may be affected by one branch of an `if-else` condition and not by the other, but the impact analysis and test selection carried upon the dependency graph will report that test as an affected test anyway.

Although our tools are modular in nature and we have tried to keep various concerns separate as far as possible, currently our tools can handle only Java bytecodes. Systems based on languages other than Java are still out of reach of our tools.

Finally, since our goal is an automated toolset, less amount of work is done in producing better displays for the outputs of the intermediate steps. Some further works can be done to create better human readable outputs for those steps.

9.3 Future Work

Most of the future work is complementary to the limitations mentioned in section 9.2. Firstly, as mentioned earlier, our tools can currently only handle *class* files. But for a real enterprise level tool, which any customer would

be ultimately expecting, changes in all the other kinds of files need to be incorporated. For several database related files like `SQL`, `PL/SQL`, `PLL` etc. we have already done some rough works. For example, to detect differences between two versions of a `PL/SQL` package file, we have written Java programs that parses two versions of the file to detect database functions, procedures, triggers etc. that have changed code inside them. We have also written Java programs with the help of regular expressions to detect differences between two `SQL` or `PL/SQL` scripts as to determine which tables in database the new version of the script may alter, update or create. We went further and also wrote Java tools to detect schema and data changes between two versions of a database table. But all these works need to be developed further to incorporate them into our toolset.

As far as the extension of dependency relation to the database is concerned, we have written a Java program that uses a tool called Java String Analyzer (JSA) (whcih itself is based on Soot [LBL+10]) to detect `SQL` string literals that are used to access the database somehow (e.g. call a database procedure) with the help of regular expressions. However, we have only tested our program on very narrow scale sample projects and we are not currently so sure about its feasibility in our real system. However, in future, this issue certainly needs to be taken care of, possibly with a better approach.

Since we store all information in a database, in future, this gives us a chance of examining the information with intelligent queries or mining the data to extract certain patterns in dependencies. The benefit of this would be that we may be able to narrow down our analysis to certain specific parts of our system.

If we are able to find some patterns in the dependencies and in the changes made by patches, in future, we might well look for doing a detailed statement level control flow or data flow analysis on specific parts of the system to take our analysis at a finer grain and to be even more accurate with the test case selection.

Many legacy systems are based on older languages like COBOL. In future, we may need to deal with such systems. And for that, our tools need to be made more generic, such as by re-factoring code and separating the concerns

even further. Also the fact that Java bytecode is a relatively higher level representation than other kinds of compiled codes (e.g. assembly, machine code) and several off-the-shelf bytecode analysis tool already exists facilitated our work to a great extent. But in the case of languages like COBOL, we would first have to write our tools for analyzing the compiled representations of these languages. This will be a challenging issue in the future.

Bibliography

[AK97] K. Abdullah and K.White. A firewall approach for the regression testing of object-oriented software. In *Proceedings of the 10th Annual Software Quality Week*, May 1997.

[AM95] Martin Alt and Florian Martin. efficient generation of efficient interprocedural analyzers with pag. In *Proceedings of the Second International Symposium on Static Analysis*, pages 33–50. Springer-Verlag, 1995.

[AOH04] Taweesup Apiwatanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 2–13, 2004.

[Api05] Taweesup Apiwatanapong. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, 2005.

[BA95] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis.* 1995.

[Bal98] Thomas Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, volume 23 issue 2, March 1998.

[BDSP04] B. Breech, A. Danalis, Stacey Shindo, and Lori Pollock. Online impact analysis via dynamic compilation technology. In *20th IEEE International Conference on Software Maintenance*, 2004.

[Ben95] K.H. Bennett. *Legacy Systems: Coping with Success.* In *IEEE Software*, volume 12, pages 19–23, January 1995.

[Bod03] Eric Bodden. Janalyzer, a visual static analyzer for Java. Technical Report 14-03, University of Kent, Computing Laboratory, July 2003. Submitted to the SET Awards, 2003.

[BRR01] John Bible, Gregg Rothermel, and David S. Rosenblum. A comparative study of coarse- and fine-grained safe regression test-selection techniques. volume 2 Issue 2, pages 149–183, April 2001.

[BS96] D.F. Bacon and P.F. Sweeney. Fast static analysis of fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications,*, volume 31 Issue 10 of ACM SIGPLAN Notices, pages 324–341. ACM Press, New York, October 1996.

[CC77] Patrick Cousot and Radhia Cousot. A unified lattice model for static analysis of a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[CCHK90] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the procedure call multigraph. In *IEEE Transactions on Software Engineering*, volume 16 Isuue :4, pages 483–487, April 1990.

[cec] *UW Cecil Group.* Electronically available at `http://www.cs.washington.edu/research/projects/cecil/`.

[Cho05] Pulak Kumar Chowdhury. Symbolic interpretation of legacy assembly language. Master's thesis, Department of Computing and Software, McMaster University, August 2005.

[Cor10] Oracle Corporation. *My Oracle Support.* 2010. Electronically available at `https://support.oracle.com/CSP/ui/flash.html`.

[CRa94] Yih-Farn Chen, D.S. Rosenblam, and Kiem-Phong Vo and. Testtube: a system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–220, May 1994.

[dif] *Jar Compare Tool.* Electronically available at `http://www.extradata.com/products/jarc/`.

[Doa07] Matthew B. Doar. *JDiff - An HTML Report of API Differences*, 2007. Electronically available at `http://javadiff.sourceforge.net/`.

[EC10] Tricia Ellis-Christensen. *What is a Software Patch.* September 2010. Electronically available at `http://www.wisegeek.com/what-is-a-software-patch.htm`.

[ecl] *Eclipse – The Eclipse Foundation open source community website.* Electronically available at `http://www.eclipse.org/`.

[EN08] Arni Einarsson and Janus Dam Nielsen. *A Survivor's Guide to Java Program Analysis with Soot.* BRICS, Department of Computer Science, University of Aarhus, Denmark, July 2008.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proceedings of the WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, May 9 2003.

[FF97] F.I.Vokolos and P.G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *3rd internatinal conference*

on on Reliability, quality and safety of software-intensive systems, pages 3–21, 1997.

[Fou10] Apache Software Foundation. *Apache Ant*, December 2010. Electronically available at `http://ant.apache.org/`.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. Acm transactions on programming languages and systems. In *The Program Dependence Graph and Its Use The program dependence graph and its use in optimization*, volume 9 Issue 3, July 1987.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 23 Issue 6, November 2001.

[GDDC97] David Grove, Greg Defouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages call graph construction in object-oriented languages call graph construction in object-oriented languages call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 32 Issue 10, October 1997.

[GHK+01] Todd L. Glaves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. volume 2 Issue 10, pages 184–208, April 2001.

[GJM03] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 2nd edition, 2003.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for Java software. In *Proceedings of the 16th ACM SIGPLAN conference*

on Object-oriented programming, systems, languages, and applications, volume 36 Issue 11, November 2001.

[HLK+97] Pei Hsia, Xiaolin Li, David Chenho Kung, Chih-Tung Hsuand Liang li, Yasufumi Toyoshima, and Cris Chen. A technique for the selective revalidation of oo software. volume 9 Issue 4, pages 217–233, July-August 1997.

[HLL+95] Mary Jean Harrold, Loren Larsen, John Lloyd, David Nedved, Melanie Page, Gregg Rothermel, Manvinder Singh, and Michael Smith. Aristotle: A system for development of program analysis based tools. In *Proceedings of the ACM 33rd Annual Southeast Conference*, pages 110–119. March 1995.

[HMR93] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, volume 18 Issue 3, June 1993.

[Hor90] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, volume 25 Issue 6, pages 234–245, June 1990.

[How98] D. Howe, editor. *Legacy System from FOLDOC - Free Online Dictionary of Computing.* August 1998. Electronically available at `http://foldoc.org/legacysystem`.

[How05] D. Howe, editor. *Patch from FOLDOC - The Free Online Dictionary Of Computing.* June 2005. Electronically available at `http://foldoc.org/patch`.

[JL94] D. Jackson and D.A. LAdd. Semantic diff: a tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, pages 243–252, September 1994.

[jvm99] *VM Spec The class File Format.* Sun Microsystems, 1999. Electronically available at `http://java.sun.com/docs/books/jvms/second_edition/html/ClassFile.doc.html`.

[KGH+94a] David Chenho Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, and Cris Chen. Firewall regression testing and software maintenance of object-oriented systems. 1994.

[KGH+94b] David Chenho Kung, Jerry Gao, Pei Hsia, F. Wen, Yasufumi Toyoshima, and Cris Chen. Change impact identification in object oriented software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 202–211, September 1994.

[KPR00] Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test application frequency. In *Proceedings of the 22nd international conference on Software engineering*, pages 126–135, June 2000.

[LBL+10] Patrick Lam, Eric Bodden, Ondrej Lhotak, Jennifer Lhotak, Feng Qian, and Laurie Hendren. *Soot: a Java Optimization Framework.* Sable Research Group, McGill University, Montreal, Canada, March 2010. Electronically available at `http://www.sable.mcgill.ca/soot/`.

[LMS97] J.P. Loyall, S.A. Mathisen, and C.P. Satterthwaite. Impact analysis and change management for avionics software. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, volume 2, pages 740–747, 1997.

[LR03a] James Law and Gregg Rothermel. Incremental dynamic impact analysis for evolving software systems. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, 2003.

[LR03b] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[LW91] H.K.N. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings of the Conference on Software Maintenance*, pages 201–208, October 1991.

[LW92] H.K.N. Leung and L. J. White. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proceedings of the Conference on Software Maintenance*, pages 262–271, November 1992.

[MPW00] Scott McFurling, Ken Pierce, and Zheng Wung. Bmat – a binary matching tool for stale profile propagation. May 2000.

[Mye86] E.W. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, volume 1 Issue 6, pages 251–266, 1986.

[OAH03] Alessandro Orso, Taweesup Apiwatanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, volume 28 Issue 5, September 2003.

[OAL+04] Alessandro Orso, Taweesup Apiwatanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, 2004.

[ora09] *Oracle E-Business Suite Patching Procedures.* November 2009. Electronically available at `http://download.oracle.com/docs/cd/B40089_10/current/acrobat/oa_patching_r12.pdf`.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings*

of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, volume 29 Issue 6, November 2004.

[PA06] S.L. Pfleeger and J.M. Atlee. *Software Engineering: Theory and Practice.* Prentice Hall, Englewood Cliffs, NJ, 2006.

[PA10] A. Passi and V. Ajvaz. *Oracle E-Business Suite Development and Extensibility Handbook.* McGraw Hill, 2010.

[Pig97] Thomas M. Pigosky. *Practical software maintenance: Best practices for managing your software investment.* Wiley Computer Publications, New York, 1997.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. volume 22 Issue 8, pages 529–551, August 1996.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. volume 6 issue 2, pages 173–210, April 1997.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. volume 24 Issue 6, pages 401–419, June 1998.

[RHD00] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. volume 10 Issue 2, pages 77–109, June 2000.

[RST+03] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, Ophelia Chesley, and Julian Dolby. Chianti: A prototype change impact analysis tool for Java. Technical Report DCS-TR-533, Rutgers University, Department of Computer Science, September 2003.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of Java programs. In *roceedings of the 19th annual ACM SIGPLAN*

conference on Object-oriented programming, systems, languages, and applications, volume 39 Issue 10, October 2004.

[Ryd79] Barbara G. Ryder. Constructing the call graph of a program. In *IEEE Transaction on Software Engineering*, volume SE-5 Issue:3, pages 216 – 226, May 1979.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35 Isuue 10, October 2000.

[Sin01] Saurabh Sinha. Interprocedural control dependence. In *ACM Transactions on Software Engineering and Methodology*, volume 10, April 2001.

[Ste] Bjarne Steensgaard. A polyvariant closure analysis with dynamic abstraction. Unpublished manuscript, 1994.

[Tes10a] Jean Tessier. *Dependency Finder*. 2010. Electronically available at `http://depfind.sourceforge.net/`.

[Tes10b] Jean Tessier. *The Dependency Finder User Manual*, November 2010. Electronically available at `http://depfind.sourceforge.net/Manual.html`.

[TM94] Richard J. Turver and Malcom Munro. An early impact analysis technique for software maintenance, January 1994.

[Whi09] Greg White. *Profile Viewer*, June 2009. Electronically available at `http://www.ulfdittmer.com/profileviewer/index.html`.

[wik10a] *Legacy System - Wikipedia, the free encyclopedia.* October 2010. Electronically available at `http://en.wikipedia.org/wiki/Legacy_system`.

[wik10b] *Patch (Computing) - Wikipedia, the free encyclopedia.* November 2010. Electronically available at `http://en.wikipedia.org/wiki/Patch_(computing)`.

[wik11] *Dynamic dispatch – Wikipedia, the free encyclopedia.* 2011. Electronically available at `http://en.wikipedia.org/wiki/Dynamic_dispatch`.