A Methodology to Support the Maintenance of Object-Oriented Systems Using Impact Analysis.

Michael Alan Hoffman  
*Louisiana State University and Agricultural & Mechanical College*

Follow this and additional works at: [https://digitalcommons.lsu.edu/gradschool_disstheses](https://digitalcommons.lsu.edu/gradschool_disstheses)

Recommended Citation  
[https://digitalcommons.lsu.edu/gradschool_disstheses/7365](https://digitalcommons.lsu.edu/gradschool_disstheses/7365)
INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
A METHODOLOGY TO SUPPORT THE MAINTENANCE OF
OBJECT-ORIENTED
SYSTEMS USING IMPACT ANALYSIS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Michael Alan Hoffman
B.A., University of New Orleans, 1978
A.S., Southeastern Louisiana University, 1983
M.S., University of Southern Mississippi, 1992
December 2000
Acknowledgements

Many people helped and encouraged me in the many years it has taken to complete this work. I would like to thank them now.

First, there is Thomas F. Higginbotham, Ph.D. He has always had a vision for my future. He persuaded me to go to graduate school. He always focused my attention on the matter at hand. Thanks for help.

Second, there is James E. Miller, Ph.D. A not-so-great undergraduate student asked him for an assistantship. By the end of my stay as USM, I was teaching a full course load and graduating. Thank you for your confidence, Dr. Miller.

Third, in time, but first in contribution, is Doris L. Carver, Ph.D. I believe there were times when she was the only person in the world that thought I could finish this degree. She has been unwavering in her professionalism. I am sure that no one has tried her patience more than I have. Thank you for not giving up on me. Thank you for all your insight, wisdom and patience. Thank you for your kindness.

For the rest of my graduate committee, Dr. Iyengar, Dr. Kraft, Dr. Skavantzos and Dr. Lax, thank you for your advice, support and patience.

My mother, Marica Thibodeaux and her husband, Louis, gave me a place to live for much of the time I was at LSU. They gave up their weekend home for me. They paid many of my bills. Although they have never set foot on the LSU campus, they have contributed greatly to this project. Thank you.

Finally, there is my wife Fei and our son Sean. They have fortified and strengthened my resolve to finish. No one has sacrificed more. No one has believed more. Thank you.
# Table of Contents

ACKNOWLEDGMENTS ................................................................. ii  
LIST OF TABLES ................................................................. v  
LIST OF FIGURES ................................................................. vi  
ABSTRACT ............................................................................ vii  

CHAPTER 1. INTRODUCTION .................................................. 1  
  1.1 Object-Oriented Systems Maintenance ............................. 1  
  1.2 Research Objectives ....................................................... 3  
  1.3 Summary ........................................................................ 4  

CHAPTER 2. BACKGROUND AND FOUNDATIONS ............... 6  
  2.1 Introduction ..................................................................... 6  
  2.2 Research Related Terminology and Lexicon .................... 6  
  2.3 Object Oriented Software ............................................... 9  
  2.4 Architectural Cliches ..................................................... 11  
  2.5 The LLSA Model .......................................................... 12  
  2.6 A Brief Introduction to Java .......................................... 13  
  2.7 Summary ........................................................................ 16  

CHAPTER 3. RELATED RESEARCH .................................... 17  
  3.1 Introduction ..................................................................... 17  
  3.2 OOTME ........................................................................... 17  
  3.3 Omega .............................................................................. 18  
  3.4 Algorithmic Analysis ..................................................... 18  
  3.5 Impact Analysis System (IAS) ....................................... 19  
  3.6 Visual Impact Analysis .................................................. 19  
  3.7 Coupling Based Impact Analysis .................................... 20  
  3.8 Comparison of Impact Analysis Approaches .................. 20  

CHAPTER 4. COMPARATIVE SOFTWARE MAINTENANCE .... 23  
  4.1 Introduction ..................................................................... 23  
  4.2 Overview of CSM .......................................................... 23  
  4.3 CSM – Why It Works ..................................................... 24  
  4.4 Content and Formalization of ELLSA ......................... 25  
  4.5 The CSM Data Model – Extended Low-Level Software Architecture .... 29  
  4.6 Extracting the “Raw” Data ............................................. 32  
  4.7 Creation of an ELLSA ..................................................... 33  
  4.8 Comparative Impact Analysis ....................................... 33  
  4.9 Instrumentation of System Components ....................... 48  
  4.10 Test Coverage Analysis .............................................. 49  
  4.11 Summary ........................................................................ 51
CHAPTER 5. PREDICTIVE IMPACT ANALYSIS .............................................. 52
  5.1 PIA Overview ....................................................................................... 52
  5.2 PIA Process .......................................................................................... 54
  5.3 Summary ............................................................................................... 58

CHAPTER 6. JFLEX - A TOOL FOR PERFORMING CSM ........................... 60
  6.1 Introduction ........................................................................................... 60
  6.2 JFlex Data Structures .......................................................................... 61
  6.3 JFlex Parsing Functions ....................................................................... 62
  6.4 ELLSA Construction and Viewing Functions ........................................ 64
  6.5 ELLSA Comparator ............................................................................. 69
  6.6 Comparative Impact Analysis and Predictive Impact Analysis .......... 71
  6.7 Code Instrumentation, Compilation and Execution .............................. 72
  6.8 Test Coverage Analysis ....................................................................... 74
  6.9 Summary ............................................................................................... 74

CHAPTER 7. VALIDATION OF THE CSM AND JFLEX ............................... 76
  7.1 Introduction ........................................................................................... 76
  7.2 Demonstrating the Feasibility of CSM, Correctness of
      ELLSA and JFlex ............................................................................. 76
  7.3 Validation of the CIA and PIA Processes and CSM ............................ 96
  7.4 Improvement of Maintenance Process .............................................. 104

CHAPTER 8. CASE STUDIES .............................................................................. 108
  8.1 Introduction ........................................................................................... 108
  8.2 Case Study 1 - Modification of a Simple Java Linked-list Database ...... 108
  8.3 Case Study 2 - Modifications to a Large Java System ......................... 120
  8.4 Summary ............................................................................................... 122

CHAPTER 9. SUMMARY, CONTRIBUTIONS, EXTENSIONS
            AND CONCLUSION ........................................................................ 126
  9.1 Summary ............................................................................................... 126
  9.2 Conclusion ............................................................................................. 127

BIBLIOGRAPHY ............................................................................................. 132

VITA ............................................................................................................. 138
# List of Tables

3 - 1 Comparison of Impact Analysis Approaches..................................................22

7 - 1 Source code for Scenario 2.................................................................78

7 - 2 ELLSA for Scenario 2.................................................................79

7 - 3 Source code for Scenario 5.................................................................81

7 - 4 ELLSA for Scenario 5.................................................................83

7 - 5 ELLSA for Scenario 10.................................................................88

7 - 6 ELLSA of original StringDemo class..............................................100

7 - 7 ELLSA for modified StringDemo class from Scenario 1.............101

7 - 8 Results of CIA from Scenario 1..................................................101

7 - 9 Partial ELLSA for original system ..............................................102

7 - 10 Partial ELLSA for modified system .............................................102

7 - 11 CAR for Scenario 3, changing a container class.........................103

7 - 12 TD for method Test.main of the modified system.........................106

7 - 13 CAR for Scenario 9.................................................................106

7 - 14 CAR for Scenario 12.................................................................107

8 - 1 ELLSA for SBD classes before modification.................................113

8 - 2 CAR for SBD original compared to SDB modified........................117

8 - 3 CAR for Case Study 2.................................................................123

9 - 1 Comparison of IA approaches including CIA and PIA..............130
List of Figures

2 - 1 The LLSA graphs ................................................................. 14

4 - 1 CSM methodology ................................................................. 24

4 - 2 Class Textual Description (TD) ............................................... 28

4 - 3 Method Textual Description (TD) ........................................... 28

4 - 4 Object Textual Description (TD) ........................................... 28

4 - 5 Contains - contained relationship ......................................... 40

4 - 6 Inherits-Contains Interaction ............................................. 43

4 - 7 Parent class using an object of a child .................................... 43

5 - 1 Affected components ............................................................. 59

6 - 1 JFlex data structures for parsing and ELLSA construction / viewing.. 61

6 - 2 ELLSA Internal Class Structure ........................................... 63

6 - 3 Algorithm for Static Data Interface ...................................... 68

6 - 4 Assigned From Interaction for Objects Algorithm .................. 68

6 - 5 Assigned From Interaction for Classes Algorithm .................. 68

6 - 6 JFlex Screen ELLSA, left pane, code, upper right, system data, bottom... 69

6 - 7 Execution trace and source code ........................................... 73

6 - 8 Measuring the test coverage .................................................. 75

8 - 1 Original GUI of SDB ............................................................. 109

8 - 2 GUI after modification .......................................................... 111

8 - 3 Test coverage analysis for Case Study 1 .................................. 112

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Abstract

Object-Oriented (OO) systems are difficult to understand due to the complex nature of the relationships that object-orientation supports. Inheritance, polymorphism, encapsulation, information hiding, aggregation, and association combine to make maintenance of OO systems difficult. Due to the presence of these characteristics in OO systems, maintenance activities on OO systems often have unexpected or unseen effects on the system. These effects can ripple through system components, complicating maintenance and testing of the system. The ability to trace the effects of maintenance provides the maintainer with knowledge that assists in debugging and testing modified and affected components.

In this research, we show that the architecture of an OO system provides an effective framework for determining the impact of system changes. We developed the Comparative Software Maintenance (CSM) methodology to support the maintenance of OO systems. Through this methodology, we model relationships and structures, analyze the models to determine components that change as a result of maintenance, and perform impact analysis to determine components that are candidates for re-testing as a result of maintenance activity. The methodology includes a new data model, called Extended Low-Level Software Architecture (ELLSA), that facilitates impact analysis. CSM locates potential side effects, ripple effects, and other effects of maintenance on class structures, methods, and objects. The comprehensive architecture model enables CSM to perform either predictive, pre-modification impact analysis or post-modification impact analysis. The improved impact analysis process found in the methodology determines impact of changes to the component level. We apply the results of impact analysis to determine
component level testing requirements. CSM enhances program understanding through the use of ELLSA. It also provides assistance for capturing complex dependencies found in object-oriented code. The methodology is implemented in JFlex. The automation provided by JFlex makes the application of CSM feasible.
Chapter 1

Introduction

1.1 Object-Oriented Systems Maintenance

Software maintenance continues to pose a challenge. Many factors hamper the maintainer's ability to do his/her job. Among the primary factors is the problem of understanding the structure and relationships that exist between the components in the software. The knowledge required for program comprehension comes from many sources, including design documentation, personal experience, knowledge of the problem domain [AHR95], knowledge of the components of the programming language(s) used to write the system [AHR95] and from observing the software in use. However, each of these factors represents only a partial picture of the system. To perform maintenance on complex systems maintainers need to know how the pieces fit together.

In this research we focus on object-oriented (OO) systems. The structure and relationships in OO systems have the potential to represent exquisite designs and streamlined implementations. Often, though, OO software presents the maintainer with a jungle of interwoven and overlapping structure and interactions. Identifying the components that need to be modified is difficult. Determining the effect of maintenance activities on the system, a tedious and time-consuming task, represents a challenging problem for the maintainer. This research helps meet the challenge by providing a system to identify the effects of changes made during the maintenance process.

The maintenance of object-oriented systems is, in some aspects, simpler than maintenance of systems developed with the procedural paradigms. Characteristics, such as encapsulation, help centralize and organize the data objects. The data structures and code that are the data objects of the system are typically bundled. But OO software has
other characteristics such as information hiding, inheritance, name polymorphism, aggregation and association that, while providing versatility and extendibility, make understanding the software a difficult task. The understanding of the structure of the code and relationships among classes, methods and objects is key to the maintenance of object-oriented software. The actual modification of the code is relatively straightforward. Knowledge of what to change and of the components affected by the change is a difficult but crucial aspect of the maintenance process.

The OO characteristics that provide powerful capabilities for implementing software systems also introduce difficulties in program maintenance. These difficulties, as described in [KUN95], are summarized here:

- The **understanding problem** is caused by encapsulation and information hiding. These characteristics cause a "delocalized plan" in which functions from several classes may be invoked to perform an operation. Functions from a called class may in turn call functions from other classes, resulting in a chain of calls. Invocation chains may become quite involved. "The implication of the invocation chains is that a tester/maintainer has to understand sequences of member functions and the semantics of the class prior to preparing any test cases and/or modifying the intended functionality. Since it is necessary to understand all the parts in sufficient detail before testing/modification, this adds tremendous complexity to testing and maintenance of OO systems" [KUN95].

- The **complex dependency problem** is caused by the presence of "inheritance, aggregation, association, template class instantiation, class nesting, dynamic object creation, member function invocation, polymorphism, and dynamic binding..."[KUN95]. These factors create complex relationships that cause dependencies between classes. The complex relationships contribute to:
1. difficulty in understanding a class in a large system
2. difficulty in determining a starting point in testing either code modifications
3. the high costs of testing
4. difficulty in discovering and testing polymorphisms and dynamic binding
5. difficulty in the identification of change impact in OO maintenance

- The tool support problem arises from a lack of OO based CASE tools. Program test sets are usually generated manually, consuming person-hours and most likely omitting certain affected code and execution paths. "Therefore, tool support is crucial in the maintenance phase." [KUN95]

In this research we developed the Comparative Software Maintenance (CSM) methodology to assist in addressing the understanding, complex dependency, and tool support problems. CSM is a process for aiding in program understanding and automating impact analysis to help reduce testing effort arising from OO software maintenance activities. Although this research investigates CSM as applied to Java, it is applicable to any object oriented language. It provides the maintainer of OO software with detailed knowledge of the interactions and relationships among class, method and object components in an OO system.

1.2 Research Objectives

The goal of this research is to develop a methodology to improve the maintenance of OO software systems. The hypothesis of this research is that the architecture of an object-oriented software system can be used to determine the impact of system changes. Through the methodology we

- model relationships and structures in OO systems
• analyze the models to determine components that change as a result of maintenance
• perform impact analysis to determine components that are candidates for retesting as a result of maintenance activity.

1.3 Summary

Maintenance of a software system is a complex and difficult task, complicated by factors such as missing or inaccurate design documentation and too few tools for learning about the system. Maintenance of object-oriented software is complicated by characteristics such as inheritance, polymorphism and dynamic binding. In this research, we use the structures and relationships that create problems for the maintainer to the maintainer’s advantage. We show that OO software low-level structure can be used to create a model of the system that aids in program understanding, change impact analysis, and program testing.

This dissertation is organized as follows. Chapter 2 provides background on the nature of the problem and the context in which the research is performed. Definitions pertinent to the research, a description of the Low-Level Software Architecture (LLSA) model that serves as the basis of the data model used in this research, and a brief overview of the Java programming language are also given. Chapter 2 also contains a discussion of the characteristics of OO software that affect maintenance. Chapter 3 provides an overview of related research. It describes several other approaches to OO maintenance. The main areas of concentration are program understanding and impact analysis.

The core of the research is described in Chapter 4. It introduces the Comparative Software Maintenance (CSM) methodology. The Extended Low-Level Software Architecture Model (ELLSA), which is used to model OO software, is introduced. The
new algorithms, data, and relationships maintained by the model are described. Comparative Impact Analysis (CIA), a major component of CSM, is presented.

The Predictive Impact Analysis (PIA) process is introduced in Chapter 5. PIA is a forward-looking variant of CIA. Chapter 6 describes JFlex, a tool for performing CSM on Java software systems.

The validation of the research and its processes is presented in Chapter 7. This chapter describes programming scenarios used to validate CSM, test the validity of the ELLSA as constructed by JFlex and validate CIA and PIA.

Chapter 8 presents extended examples to demonstrate the capabilities of CSM as implemented in JFlex. Finally, Chapter 9 presents the contributions of the research. It also describes possible future work based on this research.
Chapter 2

Background and Foundations

2.1 Introduction

There are a numerous phases in the life of a software product. The waterfall model, as presented in [GHE91], has five major phases. They are requirements analysis and specification, design and specification, coding and module testing, integration and system testing, and delivery and maintenance. This research is concerned only with the final aspect of the final phase, maintenance. The maintenance phase is the longest phase of the life cycle. Maintaining software becomes more difficult as time progresses and the system evolves.

The maintenance phase is a microcosm of the software lifecycle. Modifications to a software system require a thorough understanding of the system, integration of new requirements, development of new code, possible alterations to existing code structure, and testing. Problems often arise in the maintenance phase because of a lack of understanding of the true functionality and structure of the system. Original documentation is often incomplete or inaccurate due to previous modifications. Typically, the maintainer of the system was not a member of the original design team, but the maintainer must be as familiar with the system as the original design team. We now examine several important aspects of software maintenance that are central to this research.

2.2 Research Related Terminology and Lexicon

To facilitate a common basis of understanding for this dissertation, we define several related terms.
Software maintenance is “the modification of a software product after its delivery (to the customer) to correct errors, to improve product performance or other attributes, or to adapt the product to a modified environment” [IEEE93]. These maintenance activities are categorized, respectively, as corrective, perfective, and adaptive. [TAK96] adds preventive as a type of maintenance activity. Corrective is concerned with finding and correcting faults that have been discovered by users of the software. This type of activity usually results in a “quick fix”. The problem is isolated and repaired quickly, and then the documentation and design are changed to reflect the modification [BAS98]. Perfective maintenance is any activity performed to make the software run faster, do more, or work better. Adaptive maintenance adjusts software to function with new hardware or in a new environment. Preventive maintenance is undertaken to head off possible problems. The Y2K two-digit date fix is an example of preventive maintenance.

Reverse engineering (also known as design recovery) attempts to identify system components and their interactions and to formulate them in a higher level of abstraction [CHI90]. The goal of reverse engineering is program understanding. Reverse engineering of a software system starts with the source code of the system and works backwards to its specifications and requirements. Over the last decade much research has been conducted in reverse engineering [AIK96], [BAR98], [BENN95], [BIG89], [CG96], [KEL99], [LIN93], [NIN94] and [RAM90]. [BIG93] states that software reverse engineering is accomplished when someone can “explain the program, its structure, its behavior, its effects on operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program.” Design recovery has become an integral part of many maintenance environments [CHA98], [CHE98], [CHEN96], [HOF00], [KUN95], and [SAM90].
Reverse engineering extracts information about a software system. In many cases, the code is the only reliable source of knowledge about the system. The information recovered is of great help in the maintenance process. Reverse engineering is a pivotal part of CSM. Some of the information reverse engineering discovers is:

- system components such as functions, variables, data structures, modules, classes, and objects
- interactions and relationships between components
- lost system knowledge due to inaccurate or missing documentation
- high-level views of the software system

Reverse engineering is relevant to CSM because it starts with obtaining an understanding of an OO software system.

**Reengineering** is the process of "examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form" [CHI90]. For software systems, reengineering usually involves reverse engineering of a legacy system to extract business rules and functionality, migration to a new paradigm such as a new programming language or operating environment and the modification that migration entails, and the forward engineering of the software in the new system. CSM impact analysis is a comparison of a system with its reengineered counterpart.

**Legacy systems** are "large software systems that we don't know how to cope with but that are vital to our organization" [BENN95]. Legacy systems typically were developed using a third generation programming language but have not been maintained in a manner that could have prevented structural degradation. Many legacy systems are into their third decade of use. Lack of original structure and documentation make system
maintenance difficult and wholly dependent on the program code. The term is now
general applicable to any software system more than a few years old.

"Impact Analysis (IA) ... is the activity of identifying what to modify to accomplish a
change, or of identifying the potential consequences of a change" [ARN93]. There are
various approaches to IA including program slicing [CHEN96], code analysis [KUN94],
coupling measures [BRI99] (metrics) and tracing calls graphs and inheritance trees.

The ripple effect (RE) is the "effect caused by making a small change to a system
which affects many other parts of a system"[ARN93]. We define the ripple effect as a
phenomenon that occurs when a change in one component in a program has an effect
(typically unknown or unsuspected) on one or more other components. Ripple effect
analysis (REA) is the recursive analysis of affected components emanating from the
source of the change (both up and down) until all components involved are located and
analyzed. REA is particularly important in object programming because of the nature of
inheritance and other object characteristics.

2.3 Object Oriented Software

We now present a summary of OO concepts and how they are implemented in Java.
Object-orientation is not just a programming paradigm. It is foremost a design paradigm.
The relationships among and between components can be formed in non-sequential, non-
procedural ways. But at the same time, many low level aspects of OO languages are the
same as procedural languages. Assignment statements, conditional statements and
looping structures work the same as in procedural languages. Those structures in Java
are indistinguishable from their counterparts in C (except for inline declarations). The
properties that separate OO languages from procedural languages are encapsulation
(information hiding), inheritance, polymorphism, and dynamic binding.
Encapsulation permits the grouping of data elements (members) and methods in a declaration unit (essentially a user defined data type) called a class. The data members and methods of a class may be assigned access attributes in three levels, generally. They are private, protected, and public. Private data members are directly accessible only by methods (functions and procedures) of the class. Information hiding hides details of a class' implementation by declaring the data aspect of a class to be private. Access to the data from outside the class can be gained only by use of methods (functions) defined with the object, if it is allowed at all. The class uses some private data only internally. Private methods may be used only by methods of the class. The defining class and its subclasses may access protected data members and methods. Public data members and methods may be accessed by any declaring entity related to the class such as subclasses, container classes, and methods that create objects of the class.

A subclass is a class whose definition includes a previously defined class. Inheritance is the name of the mechanism that allows a subclass to take on the characteristics (data and code) of some previously defined class (super class or parent class). The protected or public data members and methods of the parent class are inherited by a subclass. Inheritance can cascade through many classes. Without understanding a super class, it is difficult to understand its subclasses. In multiple inheritance, a subclass may be derived from several super classes.

Polymorphism is a complex set of mechanisms that permits attributes of an object to have multiple sets of values and for operations on an object to be implemented by more than one method of the same name. Polymorphism makes static examination and testing of code difficult. Its most common form is parametric polymorphism, where a class may possess
more than one method of the same name and differentiation between them is by their parameters.

**Dynamic binding** is a means of implementing polymorphism by delaying until run time the invocation of a method to perform an operation. The method to use is determined by the number and types of arguments passed, by the use of a function pointer to a particular function, or by the type of the object in use.

These aspects of OO systems provide powerful means for implementing software systems. They also introduce difficulties in program maintenance. Research into the maintenance and structure of object-oriented software systems has been conducted by [HSI95], [DAL93], [SAM90], [WIL91], [LEJ93], [CHI94], [CHEN96], [FIO99], [KUN94], [KIR97], [RIC99] and many others. The most pertinent and relevant works are described in Chapter 3.

### 2.4 Architectural Clichés

We define *architectural clichés* as commonly occurring patterns of interactions and statements found in software systems. For example, in simple programs it is common to find “event” controlled loops that determine how much processing is done to a data set. The loop control variable is initialized, the loop control structure encountered, the data processed, and the loop control variable reassigned, ready to repeat the process. OO software has many clichés as well. Sample OO architectural cliché include:

- contains relationship – occurs when an object of one class is a data member of another. This cliché allows the creation of complex interactions between the classes, complicating program understanding and maintenance.
• uses relationship – occurs when a method of one class declares an object of another, again creating complex interactions.

• inheritance relationship – occurs when one class' base definition is derived from one or more parent classes. Single inheritance clichés are common, with only one parent class at every level of the “chain” of ancestors. Some OO languages support multiple inheritance, with multiple parents possible at every ancestor level.

• message interaction – occurs when a method of one object calls another method.

2.5 The LLSA Model

The research described in this dissertation builds on a model of object-oriented software given in [SHR96]. The model is called the low-level software architecture model (LLSA). LLSA is an abstract view that describes the physical and logical dependencies between software components. It consists of textual descriptions of software components along with their interface and interactions with other components. The LLSA can be represented by graphs that display several relationships between the various components of an OO software system. The basic LLSA graph describes three distinct OO concepts: classes, functions, and objects. The graph describes component interfaces and interactions.

Three possible views of the system can be constructed from the LLSA graph. First is the control flow graph. It is obtained by performing the transitive closure operation on the interactions among components described in the LLSA graph. Second is the component domain graph view that is a restricted view of nodes of the LLSA graph that belong to a particular domain such as classes. A subgraph of each domain and its interactions between components is rendered. The third view is the rooted component subgraph view. This view selects one component and describes all the interactions and dependencies for the root.
component. The LLSA of a system is obtained completely by the set of rooted subgraphs for all components. Figure 2 - 1 shows a control-flow graph, a component graph view, an LLSA, and a rooted component subgraph.

In [SHR96], fundamental patterns are defined as “the syntactic constructs of an object-oriented language that correspond to the LLSA interactions”. The fundamental patterns concept is also referred to as architectural clichés in the literature. Architectural clichés cover a wider range of interactions, including non-OO aspects such as loop structures and logic structures. In this research, we refer to fundamental patterns as architectural clichés.

The content of a rooted component subgraph can be expressed as a textual description. A modified version of the textual description is employed in CSM. The textual representation could be visualized as a collapsible tree structure with infinite levels. This characteristic makes this structure ideal for representing variable length relationships such as call graphs or inheritance trees.

The LLSA model can be extracted through the use of pulse[SHR96]. It is a prototype tool that operates on C++ code. In this research, we developed a software tool called JFlex that extracts an extended version of the LLSA (the ELLSA) from Java systems.

2.6 A Brief Introduction to Java

Java is object oriented programming language that was developed by Sun Microsystems. Java, officially introduced in May 1995, was designed to be secure, safe and portable. Its most significant break with traditional languages is in its execution. Most traditional languages are compiled to a machine dependent executable form. Java is compiled to an intermediate form know as byte code. Byte code is then executed by a “virtual machine”, a runtime environment for Java native to the host machine. The virtual
Figure 2-1 The LLSA graphs [SHR96]
(a) Complete LLSA view of a system
(b) Control Flow View maps control paths
(c) Component Graph View shows interactions among like components
(d) Component Subgraph Rooted at C4 shows all interactions on a single component
machine concept allows any Java program written on any platform to execute on any other platform with a Java virtual machine.

Java programs are either applications or applets. An application is a traditional stand-alone program. An applet is a program that executes under the control of another program. Web browsers use applets mainly as means of introducing sophisticated controls, images, and manipulation into Web pages.

All aspects of a Java program are members of either a class or an interface. A class is a construct for defining a collective group of data and methods that operate on the data. It is essentially a user defined data type that may contain data and actions on that data. All methods are contained within a class. All data (variables and objects) are defined either within a method or as a class data member. A Java program uses a class by instantiating an object of that class.

An interface defines data and methods that are usable directly by another class without an instantiated object. Interfaces are implemented by a using class.

Java supports class inheritance. The Java keyword extends signifies inheritance. For example,

```java
class student extends person
```
declares a class named student that inherits protected and public members from another class named person.

Classes and individual members can have public, private or protected access as defined in section 2.3. Java has other "modifiers" such as final, static and abstract. A final member is a constant. A final method cannot be overridden. A static member or method exists without an instantiated object. Static members and methods are class wide, accessible to all objects of that class and to any class sharing their directory. An abstract
method must be overridden. An abstract class cannot be instantiated as an object but can be extended.

All sub-routines, called methods, are part of a class. There are no "global" methods or data, although using static methods and data in a limited scope can create the effect. The standard start method "main" is static. Java assumes the presence of main, just as in C and C++.

2.7 Summary

In this chapter, we presented background and foundational information in several areas pertinent to this research. We defined terminology relevant to the research, described factors that complicate understanding OO software systems, and defined architectural clichés. The chapter includes a description the LLSA model and the three views of an OO system that the model supports, as well as a brief overview of Java.
Chapter 3

Related Research

3.1 Introduction

Research related to this work comes primarily from the areas of impact analysis and software maintenance environments. Other related, but less significant, areas include design recovery, architectural design, and design patterns.

3.2 OOTME

In [KUN94], [KUN95] the various types of code changes that occur with OO software systems are presented. Changes are classified as data, class, method or class library changes. Rules, based mainly in graph and set theory, are presented for detecting changes in OO components and for detecting further affected components. Change impact identification is achieved through a prototype tool called the object-oriented test model environment (OOTME). OOTME implements the rules for change types and identification. A method for class firewall construction is also presented. A class firewall is the set of affected classes produced in an OO system whenever modifications are made to the system. OOTME provides a graphical representation of a OO system that is similar to the LLSA. A model consisting of three diagrams that define OO systems is presented. The object relation diagram (ORD) represents relations among classes such as inheritance, instantiation, uses and others. The block branch diagram describes control structure for member functions and interfaces to other functions. Its main uses are to derive test cases for functions and derive data dependence relations across functions and objects. The object state diagram (OSD) represents an object's state behavior. The OSD is an aggregate of state machines for each state dependent data member of a class. The OSD models the inheritance and aggregation of
object state behavior. These diagrams combine to generate an overall picture of an OO system, a picture that is created by the OOTME. The current main use of the OOTME is to design optimal test suites for OO software.

3.3 Omega

In [CHEN96], an integrated environment for C++ program maintenance is presented. The paper describes three "new" dependence graphs specific to OO systems: message, class and declaration dependence in a model called C++DG. Additionally, several new slicing techniques are presented. The use of the new dependencies and slicing on code maintenance is described, specifically as to the ripple effect and regression testing. The dependencies described are similar to the dependence problems described in [SHR95]. The application of the discovered dependencies and program slicing leads to recursive analysis of the ripple effect caused by code modification. As the effects are located, classes and methods affected can be "marked" for testing or re-execution in the testing phase.

3.4 Algorithmic Analysis

In [LI96], four algorithms are presented that measure the effect of proposed changes to OO systems. The ripple effect is calculated by application of algorithms that:

1. calculate the change effects inside of a class
2. calculate the change effects among clients
3. calculate the change effects among subclasses
4. measure the total effect by driving the algorithms in 1,2 and 3

[LI96] presents details of how different types of changes affect the system. Changes are broadly categorized as method or member change, and then refined to more detail such as adding a member or changing an attribute.
The algorithms calculate the transitive closure of each of the potentially affected classes and methods, as in [SHR96]. With the added information that [SHR96] provides, it will be possible to greatly improve upon the information provided by the algorithms in [26]. Recognition of low-level design patterns, effects of data type changes, and effects of addition and deletion of classes can be drawn from the LLSA model of a OO system.

3.5 Impact Analysis System (IAS)

IAS [BARI95] is a system for performing high-level impact analysis. IAS appears to be primarily concerned with function call graph dependencies. This approach is “based on modeling of both the dependencies within the maintained software system and the way modifications induce ripple effects.” The software system being maintained is modeled as a set of typed objects such as code modules, functions, design objects and test cases linked by various dependency links such as documentation, composition, and version links. These objects form classes (not in the OO sense) of relations such as Requirements, Validation-Test-Cases, Is-Tested-By, and Is-Composed-Of.

Changes to the software system also are modeled as types and links. Objects such as functions are modified and produce effects that are modeled by links such as is_called_by. Links are followed from object to object. IAS records “impacts” as they result from “propagation rules”. A software tool implementing IAS was produced that provides a high level view of proposed changes to a software system.

3.6 Visual Impact Analysis

[GAL96] employs program slicing to select a point in an ANSI C program for observation. The method looks at program variables and essentially models dependencies that exist among variables via assignment statements and parameter passing. The method is a visualization of the data collected by the Surgeon’s Assistant and is called the
Decomposition Slice Display System [GAL96]. A variable's decomposition slice is the set of all statements in the program that contribute to the variable's computation. In [HUT98] Visual Impact Analysis is improved through the recognition of further dependencies such as interference, a list of what other variables could "interfere" with the maintenance of some other variable. The display is improved to provide more detail of the dependencies.

3.7 Coupling Based Impact Analysis

[BRI99] presents the results of an empirical study that uses change history to infer a causal link between coupling in an OO system and impact analysis. The data, drawn from an OO system called LALO, contains information on relationships among the 90 classes of the C++ system. New coupling measures are introduced: static method invocations, polymorphically invoked methods, and direct and indirect aggregation relationships. Statistical analysis is performed on the change data using univariate regression analysis on about 18 separate metrics that measure coupling. The authors claim that the coupling measures developed in this approach are "good indicators of ripple effects". The coupling based model can indicate class pairs with a high probability of ripple effects. This approach is predictive at a coarse level of granularity.

3.8 Comparison of Impact Analysis Approaches

Table 3 - 1 visualizes the major aspects of the various IA approaches described in this section and compares those works with the results of this research, CIA and PIA. The table's contents are described now.

- The first column is the name of the approach or method.
- The second column names the data representation model used for the approach.

Some of the methods listed in the table do not described a "model", but a process.
Storage and representation of the extracted data is left to the maintainer to model, if desired. For example, Algorithmic IA is a set of rules for IA.

- The third column reports the programming language(s) upon which the approach is employed.
- The column headed Automated may contain Yes, No or Semi. Semi means that IA data are extracted from the code being examined, but that the impacts listed may possibly not be real. That is, some data generated by the approach represents categories of possible impacts. It is left to the maintainer to ascertain the correctness of the approach.

- Granularity refers to the type of components being reported based upon the program elements being examined. For example, if the approach uses high-level modules and design documentation, it has coarse granularity. Medium granularity is claimed for systems using only generic descriptions of interactions among classes. For example, it is known that a classes uses another class, however, there are no details stating the level of interaction. If method variables and parameters are examined, then we would say the approach has fine granularity. Multiple refers to a mixture of levels of granularity.
- Predictive indicates if the approach can forecast affected components.
- Predictive Detail indicates a measure similar to granularity. Low means it reports only class relationships such as inheritance and aggregation. High means the approach reports class, method and object relationships that are affected or participating in the modification of the program.
- Comparative states if the approach can compare two sets of code, pre- and post-modification, to perform post-modification impact analysis.
- Display limit refers to the number of components that can be reasonable presented in the output of the approach, if applicable. Most are box and arrow drawing with limited presentation power.

- The last column lists the source of the IA.

Table 3 – 1 Comparison of Impact Analysis Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Model</th>
<th>Automated?</th>
<th>Granularity</th>
<th>Predictive</th>
<th>Comparative</th>
<th>Display Limits</th>
<th>Analysis Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOTME</td>
<td>ORD</td>
<td>Semi</td>
<td>Medium</td>
<td>Yes</td>
<td>Low</td>
<td>Yes</td>
<td>Low</td>
</tr>
<tr>
<td>Omega C++DG</td>
<td>No</td>
<td>No</td>
<td>Multiple</td>
<td>No</td>
<td>No</td>
<td>NA</td>
<td>Low</td>
</tr>
<tr>
<td>Algorithmic IA</td>
<td>NA</td>
<td>No</td>
<td>Fine</td>
<td>No</td>
<td>NA</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>IAS</td>
<td>Ukwn</td>
<td>Semi</td>
<td>Coarse</td>
<td>Yes</td>
<td>Low</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>Visual IA</td>
<td>Ukwn</td>
<td>No</td>
<td>Coarse</td>
<td>Yes</td>
<td>Low</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>Coupling IA</td>
<td>NA</td>
<td>No</td>
<td>Coarse</td>
<td>Yes</td>
<td>Low</td>
<td>NA</td>
<td>None</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Chapter 4

Comparative Software Maintenance

4.1 Introduction

Comparative Software Maintenance (CSM) is a methodology that models OO software relationships. It locates changes made to an OO software system as a result of maintenance, and predicts or determines the affects produced by the changes, facilitating testing the of an OO system by locating affected components.

4.2 Overview of CSM

CSM is a multi-stage maintenance methodology model that encompasses the following:

- determines OO system components – classes, methods and objects, and interactions such as aggregation, inheritance and uses
- models system components in the Extended Low-Level Software Architecture (ELLSA) for the OO systems
- creates “virtual” software systems for use in PIA
- compares ELLSA model structures for PIA and CIA in order to determine which components will be or have been affected by changes to the OO software system
- instruments source code for testing execution coverage and tracking dynamic execution data

Figure 4 – 1 shows an overview of CSM. CSM performs static and dynamic analysis of OO software systems. CSM provides a textual description (TD) of an OO system. CSM discovers relationships such as aggregation, inheritance, association and polymorphism, and other OO characteristics and displays them in a TD. The TDs form a view for the...
whole system that presents its interactions and relationships in a unified model. CSM performs detailed change analysis of OO software systems. The result of change analysis is the basis of the impact analysis performed by CSM. CSM helps automate the testing process by automatic instrumentation of modified source code and test coverage tracking.

This chapter describes CSM components and the ELLSA model. Section 4.4 describes the ELLSA. Sections 4.5 and 4.6 describe ELLSA generation. Section 4.7 describes ELLSA comparison and impact analysis.

4.3 CSM – Why It Works

OO software systems possess certain characteristics such as inheritance, aggregation, association and polymorphism that contribute to a measurable and recognizable set of relationships that can be used to aid in its own maintenance. These relationships form links between components in the system. A component is “an architectural element or design module having an interface.”[SHR96]
The types of architectural elements in OO software are consistent enough in form and function to be architectural clichés. The architectural clichés such as inheritance, aggregation, uses and association are indicators of the type of links among components one should expect to find in the system, and therefore of the types of links and relationships that will be affected when one component in the system is modified due to maintenance activities.

CSM takes advantage of the structure of an object oriented software system. CSM maps dependencies in the system. The dependencies, combined with an understanding of the structure of OO architectural clichés, enable CSM to perform software change impact analysis in order to identify affected components. The determination of affected components allows the maintainer to concentrate his/her testing on impacted components.

It should be noted that this research is not concerned with correcting the impacted components, only reporting them. We assume that the original system is free of syntax, semantic, and linkage errors. We also assume that the changes themselves are free of syntax, semantic, and linkage errors. CSM does not check for component declaration or instantiation. For example, if a method call is added to a method, CSM assumes that the called method exists.

4.4 Content and Formalization of ELLSA

We define textual descriptions for the three components of the ELLSA – class, method and object. We describe the content of the ELLSA.

Informally, a textual description (TD) is a collection descriptive data about a component in an OO system. A TD contains information about interactions engaged in by the component. For classes, a TD contains a description of the class’ data members, a listing of the class’ methods, and a description of the relationships and interactions a class
has with other classes. For methods, a TD contains a description of parameters and local variables of a method, object components contained in the method and interactions with other methods. For objects, a TD contains data dependencies with other objects and usage description. TDs for all components enumerate the interfaces for the component. An interface is a description of how a component can interact with other components.

[SHR96] refers to the textual descriptions as component description templates. The class component TD duplicates some of the data found in both the object and method component descriptions. The duplication helps speed construction of the ELLSA. As an example of data duplication, a class needs to know the names of its methods. There may be dozens of methods listed for a class. The method, though, has only one class. Each method may create objects of some class. The textual description of the class declaring the method must list the classes created by all of its methods. This list results in duplication in the output, but it facilitates comprehension of the structures and interactions of the system.

Figures 4–2, 4–3, and 4–4 present the contents of the ELLSA.

The ELLSA model stores much of the structure, and many of the relationships and interactions that exist in an object-oriented system. CSM uses this knowledge of the system to determine the impact of changes to the system.

The class domain (CD) is the set of all class TDs in an OO system. The interactions among class components are present in the CD. The method domain (MD) is the set of all method TDs in an OO system. The interactions among method components are present in the MD. The object domain (OD) is the set of all object TDs in an OO system. The interactions among objects and between an object and the rest of the system are present in the OD.
4.4.1 Formalization of ELLSA

We formally define the ELLSA and its components as follows:

Given

\[ P = \text{object-oriented software program} \]

\[ c_i = \text{a class in } P \]

\[ \text{CTD} = \text{a class textual description} \]

\[ \text{CD} = \text{class domain} \]

\[ m_i = \text{a method in } P \]

\[ \text{MTD} = \text{a method textual description} \]

\[ \text{MD} = \text{method domain} \]

\[ o_j = \text{an object in } P \]

\[ \text{OTD} = \text{an object textual description} \]

\[ \text{OD} = \text{object domain} \]

\[ I(x) = \text{the set of interactions maintained on class, method and object components} \]

\[ D(c) = \text{the data member interface of a class} \]

\[ M(c) = \text{method interface of a class} \]

\[ O(m) = \text{the set of objects created by a method} \]

\[ C(m) = \text{the call chain of a method} \]

\[ \text{DD}(o) = \text{the data dependencies of an object in } P \]

then

Class Textual Description (CTD) for a class \( c \) is

\[ \text{CTD}(c) = \langle I(c) \land D(c) \land M(c) \rangle \]

\( \text{CD} \) is as:

\[ \text{CD}(P) = \bigcup \text{CTD}(c_i) \mid \text{for } \forall c_i \in P \]

27
Name: name of the class
Location: file where the class declaration is stored
Data Interface: names, data types and attributes of all data members
Static Data Interface: public and protected data members of ancestor classes
Static Method Interface: public method members, with parameters and local declarations
Object Family: all objects of that class type and associated locations
Descendents: all subclasses of the class
Ancestors: all parent, grandparent, etc., classes of the class
Container classes: all classes that contain instances of the class as a declared data member
Contained classes: all classes that are the data type of a data member in the class
Create object classes: all classes whose objects are created by methods of the class as local variables or as parameters
Created by classes: all classes that create instances of the class in one or more of their methods as local variables or as parameters
Dynamic interface: public methods inherited from ancestors and not overridden
Calls methods: methods called by member method of the class
Assigned from: classes upon which the class has a data dependency via assignment statements

Figure 4 – 2 Class Textual Description (TD)

Name: the name of the method
Location: file where the method declaration is stored
Creates objects: all classes created within the method
Static interface: classes created when a constructor is a parameter’s data type
Dynamic interface: descendant classes of classes in the static interface
Calls: methods called by the method
Used by: classes that have methods call the method
Called by: methods that call the method

Figure 4 – 3 Method Textual Description (TD)

Name: name of the object
Location: the file in which it is declared
Declared by: method declaring the object
Static type: class type
Static interface: member methods of the class of the object
Actual interface: methods that are called by the object
Dynamic type: names of descendant classes of the static type
Assigned from: data dependencies from assignment statements for the object

Figure 4 – 4 Object Textual Description (TD)
Method Textual Description (MTD) for a method \( m \) is

\[
\text{MTD}(m) = <I(m) \wedge O(m) \wedge C(m)>
\]

The method domain (MD) is

\[
\text{MD}(P) = \bigcup \text{MTD}(m_i) \mid \text{for } \forall m_i \in P
\]

Object Textual Description (OTD) for an object \( o \) is

\[
\text{OTD}(o) = <I(o) \wedge DD(o)>
\]

The object domain is

\[
\text{OD}(P) = \bigcup \text{OTD}(o_i) \mid \text{for } \forall o_i \in P
\]

The extended low-level software architecture of \( P \) is

\[
\text{ELLSA}(P) = <\text{CD}(P), \text{MD}(P), \text{OD}(P)>
\]

4.5 The CSM Data Model – Extended Low-Level Software Architecture

In order to store and represent the architectural clichés of an OO system and other program data, we define a data representation model. The data model is capable of capturing and storing information about individual internal class structure, external relationships with other classes, and the structure of method interfaces. Knowledge of local and parameter objects, call chains, uses and used by relationships, and information about the object instances of an object system must be recorded as well.

The LLSA model [SHR96] serves as the basis for the Extended Low-Level Software Architecture (ELLSA) model. LLSA is described in Chapter 2. Because the LLSA lacks certain data and relationships required by CSM, we extend the LLSA for use in CSM. Sections 4.5.1 and 4.5.2 describe extension of the LLSA model.

4.5.1 LLSA Data Extension

Maintenance activities on a class consist basically of modifying either data members or member methods. Data members can be modified by
• addition of members
• deletion of members
• change to the access attributes (public, private, protected)
• change to a type or aggregate used to define a data member
• change in data value through assignment (data dependency)

Each of these changes can have an impact on the structure and relationships in the system. In order to track the impact, the model must maintain data about the data members. The minimum information required to perform static impact analysis is

• data member name
• data member data type
• data member access attributes (may be more than one – static final public int x; is perfectly good Java)

The textual description of LLSA components does not account for data members of a class. The LLSA model could not indicate that a change had occurred to a data member. In order to provide this capability in the LLSA model, we extended it to include the minimum data member information required: data member name, data type and access attribute.

4.5.2. LLSA Interaction Extensions

Sections 4.5.2.1 and 4.5.2.2 describe data dependency interactions and the static data interface relationship of a class.

4.5.2.1 Data Dependency Interactions

CSM requires data dependency information in order to perform impact analysis. It is common to find situations such as this:

X is data member of class A, Y is a data member of class B.
A.X = B.Y is statement in the program.

There is certainly a data dependency of X on Y. Furthermore, there is a dependency of class A on class B. The assignment statement may only change the value of the data member, but in changing the value of a data member, the state of the object to which the data member belongs is affected and possibly altered.

Two interactions called assigned from for object and assigned from for classes are added to the LLSA model to record the data dependencies of objects and classes. The assigned from interaction for objects records the names of qualified data members of object instances and methods that participate in any expression that assigns a value to the object. For example, from A.X = B.Y, B.Y is an element in assigned from for A. In the assigned from interaction for class A, B is an element. All data members of a class participate in the creation of the class' assigned from interaction. The algorithms are presented in section 6.4.1.4.

4.5.2.2 Static Data Interface of a Class

The static data interface (SDI) of a class is the union of the non-private data members of the class' ancestor classes and the public and protected data members of the class. Overridden data members from ancestor classes are excluded from the subclasses' static data interface. If a class has no ancestor or all the data members of its ancestors are private, the static data interface of the class is its data members. The SDI provides the maintainer with a complete list of data members that may be accessed by any client class or method of this class. The algorithm is presented in section 6.4.1.3.

4.5.2.3 Method Data Extension

As with classes, maintenance activities on methods require the inclusion of additional data in the model. The LLSA notes local variables within methods only if they are
objects. In the ELLSA, all local variables’ names, types and attributes are recorded. This helps determine if any new variables were added to a method that would cause it to need to be retested.

4.5.2.4 The Actual Interface of an Object

The static interface of an object component does not reveal what messages are actually sent. The Static Interface lists all the potential messages, but many of them are not used within the scope of the object declaration. The Actual Interface (AIO) of an object describes the messages that are actually sent by the object. The AIO of an object O is

\[
\text{AIO} = \bigcup O.\text{message} \text{ such that } (O.\text{message is sent AND } O.\text{message } \notin \text{ Static Interface of } O)
\]

The AIO is a listing of all messages sent to the object from the object’s Static Interface. It helps reveal how the method interacts with the object. The AIO is determined by examining the \textit{calls-methods} interaction of the method that contains the declaration of the object. The ELLSA notes and records all calls involving object O.

4.6 Extraction of the “Raw” Data

CSM constructs the ELLSA of an OO system in two major phases. In the first phase, the system code is parsed in order to locate and extract the obvious “raw” data and relationships such as class names, class method name, and parameter lists. The second phase consists of the application of algorithms to the extracted “raw” data in order to discover the relationships that are not obvious, such as descendent classes, used by, and called by.

The data extracted from a class declaration are:

- class name
- extends name or parent class (if any)
• implements name(s) (if any) (a type of multiple inheritance)
• names, types and attributes of data members
• names, return types, parameters and local variables of class methods
• method invocations
• assignment statement level data dependencies

All other data and interactions presented in the ELLSA are created, compiled or elucidated from the raw data.

4.7 Creation of ELLSA

Generation of the ELLSA takes place after extraction of the raw data. For each class, method and object in the system a set of algorithms operates on the raw data. The algorithms are applied first to classes, then methods and finally objects. The reasoning behind this ordering is that much of the information created for methods and objects can (or must) be extracted from information already produced for the classes. For example, the static interface for a method is composed of the classes that occur in the method's parameter list. Those classes must be determined first. The dynamic interface of a method consists of the static interface of its descendant classes, which again must be known in order to produce the method TD. A discussion of the algorithms that produce the ELLSA is presented in Chapter 6.

4.8 Comparative Impact Analysis

CSM performs change analysis on a modified Java system, and then exploits the detailed knowledge of the system stored in the ELLSA to perform impact analysis. The goal of impact analysis is to identify those components that are affected by a change to the system. Comparative Impact Analysis (CIA) is the core of CSM. CIA uses the low
level architecture and architectural clichés of an OO system to determine the nature of changes to the system and to determine side effects those changes might produce.

How is impact analysis done? The central idea of CIA is based on a "snapshot" concept. We take a snapshot of the structure of an OO software system before any maintenance activity is applied. After modifications to the system, however simple or involved, we take another snapshot of the system. We then compare the two views. The views are the ELLSA of the Java software system. Why do we want to compare them? By examining certain aspects of the two systems, we can determine which classes, methods and objects will be or have been affected by maintenance activities. With the detailed knowledge of the system in the ELLSA, we can follow the ripple effect created by changing class structures until all affected classes, methods and objects have been identified. In the large, CIA is the set difference operation applied to two software architectures. Let P1 and P2 be two versions of the same software system at two different points in time. Then,

\[
CIA = \text{ELLSA}(P1) - \text{ELLSA}(P2) \cup \text{ELLSA}(P2) - \text{ELLSA}(P1)
\]

The result of CIA is a list of classes, methods and objects affected by the changes to the system.

4.8.1 Types of Changes

Changes can be made to classes, methods, and objects or to the system as a whole. Systemic changes in an OO system consist of addition or deletion of classes or interfaces.

Many types of changes can ripple through the system and have potential impact on the system. Simple changes are listed in sections 4.8.1.1 through 4.8.1.3 Changes to component interactions can be either simple (addition or deletion of a method invocation)
or complex (addition of a *uses/used by* interaction between classes). A complex change contains one or more potential ripple effects. Complex changes are changes to low-level architectural clichés. Complex changes discovered by CIA that cause the inclusion of a component in the list of changed/affected components include are discussed in sections 4.8.3.1.1 through 4.8.3.3.

4.8.1.1 Changes to Class Structure

Classes are essentially data and methods. The maintainer can apply the following types of changes to classes:

- add a data member
- delete a data member
- modify access attributes
- add a method member
- delete a method member
- change a method member

Some previous research [KUN94], [LUI96] has listed changing a data member's data type as a type of change that can be applied to a class. Data type "changes" are not possible. In our view, changing a data type effectively renames the data member. In Java, there are two possible outcomes of changing a data type. 1) The change may be ignored because the data member is used in such a way as to allow for implicit conversion of the value. In this case where is the impact of this change? What is the affect? The structure remains unchanged and the data dependency is unchanged. 2) The change will cause a syntax error because the data member type is illegal in the context. The data type explicitly defines the *context* in which a variable can be used. Changing the data type changes part of what defines a variable: the context in which it is allowed to operate.
legally. This is the same as deleting the variable and adding a new variable with the same name but different data type.

The actions of adding or deleting a data member seem to require no explanation. However, these actions are not as simple and straightforward as they first appear. Data members, if of primitive type, can appear and disappear with minimal impact to the system. Of course, dependant classes and methods require recompilation and testing. Data dependencies may be affected as well. Nevertheless, at the architectural level, changes to primitive types do not typically engender effects. If the data member is an instance of a class then there is an affect on the architectural structure of the system. The addition of a data member that is an instance of a class creates contains and contained by interactions at the very least. The data member’s methods are likely to be invoked as well, creating uses and used by interactions between the methods of the two classes. The modification of access attributes can change the way in which a data member is viewed by the system. For example, if name is a public member of class employee, it can be accessed by simply referencing it as a qualified member of an instance of class employee, as in e.name, where e is an object of class employee. If the access attribute is changed to private, references such as e.name become illegal. This type of change to a data member impacts program behavior.

Adding a method potentially affects the class containing the method; all descendant, aggregate and associate classes of the changed class; and objects of that class.

4.8.1.2 Changes to Methods

A change to a member method involves one or more of the following actions:

- add a function call
- delete a function call
• add a locally created object instance or variable
• delete a locally created object instance or variable

We consider adding or deleting a parameter that is an object instance as actually adding or deleting a method. Java and C++ support name polymorphism on methods, which means that the only uniquely identifying property of polymorphic method names is the parameter list. If the parameter list is modified, the unique footprint that describes the method is deleted and a new one created. This modification effectively renames the method, which is no different from deleting a method and adding a new method.

4.8.1.3 Changes to Objects

Object modifications are few. An object instance can be added or deleted in a parameter list of a method or as a local variable. Depending on the programming language, the scope of the instance can be changed by relocating the object's declaration statement. We consider the addition or deletion of objects as class data members to be a modification to a class. Modifying the right-hand side of an assignment statement in which an object is the variable being assigned can change the data dependency of the object. This type of change affects data dependencies for the object and its class.

4.8.2 ELLSA Comparison

The comparison of the ELLSAs yields a list of all system components that are affected by the maintenance activity. The comparison is performed in the following order: methods, classes, object instances. The basic maintenance activities described in 4.8.1 trigger the automatic inclusion of a class, method or object in a list of simple modifications. From this list CIA determines all components of the system that will be
affected by these simple changes described in section 4.8.1 using the ELLSA of original and modified systems. Chapter 6 presents a technical description of ELLSA comparison.

4.8.3 Impact Identification

The impacts and ripple effects of the changes can be traced after the list of simple modifications has been compiled. Ripple effect and ripple effect analysis (REA) are defined in Chapter 2. The trace is accomplished by using the knowledge of the system accumulated by the ELLSA. Sections 4.8.3.1 and 4.8.3.2 describe the types of impacts identifiable.

IA can be as simple as determining that a change causes no impact. For example, the deletion of a method will cause it to be included in the modified components list. This method was defined in a class but is never invoked by any component in the system. The defining class has no subclasses. In other approaches to REA, the deletion of this method would register impacts in the defining class and any contained or used interactions. CIA is able to make the determination that this change will not impact the system.

4.8.3.1 Complex Change Class Modification Impact Analysis

The effect of modifying a class is varied. Changes can ripple down the inheritance tree, through uses and used by relationships, through container and contains relationships or in combination. If a class modification has no effect on any component in the system then the class should be removed because it is not being used!

4.8.3.1.1 Container Class Modification

A container class has as a data member an instance of this class. The container class is subject to all the simple changes listed in 4.8.1. A modification to a container class either strengthens or weakens bonds to this class. The container class already has access to the public members and methods of this class. Modifications that will affect the overall
architecture of the system and the direct interactions between container and contained are:

- using more or less of this class's methods thereby altering the call graph
- accessing more or less of this class's data members thereby increasing or decreasing data dependencies
- deletion of the contains interaction by deletion of the object instance of this class

All other modifications to the container have no effect on this class, provided that contains/contained by were the only interactions. Figure 4 – 5 shows the contains and contained by interactions.

4.8.3.1.2 Contained Class Modification

A contained class is one whose instance is a data member of this class. Modifying a contained class introduces a variety of impacts into the architectural structure to which the modified class belongs. The contained class is subject to all the simple changes listed in 4.8.1. The nature of the change(s) must be determined before IA of this class can be accurately performed. For example, suppose that a public data member is deleted from a class contained in this class. What if the deleted data member is not used by this class? There is no impact to this class in this case. Testing this class would be unnecessary. Similar circumstances exist when deleting or modifying an unused public method or modifying an unused public data member.

The modification of a used data member will either change the data type or change the scope. As described in [LI96], scope changes from private to protected/public or from protected to public increase visibility. These have no effect on references. Changes from protected to private or from public to protected/private decrease visibility, thereby decreasing access. In a contained class cliché, this type of change will cause an impact. In
fact, it will cause a syntax error upon compilation of the code. Changes to access attributes of methods result in similar circumstances for methods. We do not consider a change to the data type of a data member as a viable change, as argued in section 4.8.1.1.

Modification of a used method may or may not affect its client. If the signature of the method is changed, calls to this method will no longer match. The containing class’ methods will be impacted. Again, as in the case of data members that are changed, this type of change will cause a syntax error upon compilation of the code. As we described in section 4.8.1.2 signature changes are not possible. A change to its signature effectively renames a method. The impact is that of deleting the method.

```java
class Person {
   //Person contains Employee
   private String Name;
   private Employee emp; //I'm contained
   Person (String n, int id, boolean p)
   {  emp = new Employee(id, p);
      Name = n;
   }
}

class Employee {
   protected int ID;
   protected boolean permanent;
   Employee(int id, boolean p)
   {  ID = id;
      permanent = p;
   }
}
```

Figure 4–5 Contains / contained relationship

Changes in the body of a method belonging to a contained class that effect this class are the addition of a function call and the deletion a function call. These changes alter the call chain stemming from methods of this class.
Deletion of a used data member or method, of course, impacts the using method and class. Without modification to the remaining code, there will be syntax errors or linkage error upon compilation of the code.

The impacts described in this section are the impacts that occur to this class when the presented changes are made to a contained class. Addition or deletion of occurrences of locally defined objects has no effect on this class. The only other manner in which this class could be affected is if an instance of itself is passed as a parameter to a method of the contained class - a poor OO programming practice.

4.8.3.1.3 Assignment of a New Ancestor Class to an Orphan Descendent Class

If a class does not have a parent, then it is a contained class, a used class or a class that has no usage at all. In the latter case, there can be no impacts from assignment of a new ancestor. This is not to say that new interactions are not available; they almost certainly will be. However, the act of merely making interactions available does not imply that they are used.

For contains or contained interactions, impacts occur if the parent contains the descendent, thereby setting up a recursive declaration. This situation will inevitably cause a run time error by causing the stack to overflow. Figure 4 - 6 demonstrates this condition. If the child contains an instance of the parent there will be duplication of the public and protected members of the parent class. Any member function of the child will have access to the protected/public inherited members and the protected/public member from the contained parent class instance. This situation lends itself to misunderstanding, confusion, and error.

For used/uses, if the parent class has an instance in a method of the child class there will be at least duplication of the public/protected data and methods of the parent in that
method. The duplication includes private members in situations in which the descendent class has a used by interaction with a method of the parent class. Figure 4 – 7 presents Java code typifying this case. The discovery and documentation of these types of interactions and relationships are improvements over previous IA approaches because CIA not only tells the maintainer what is affected but also why it is affected.

4.8.3.1.4 Assignment of a New Ancestor Class to a Descendent Class

It is natural to assume that a child class tends to avail itself of a great deal of the functionality and structure of the parent class. The impact resulting from assigning a new ancestor may be tremendous or they may be non-existent. The level of impact depends on the relationships the class currently has with its parent class, if it has any at all. An analysis based upon actual relationships and interactions is required to accurately determine the impact. Providing the analysis is one of the major improvements provided by this research. CIA records details of the interactions between the parent(s) and child classes. It is possible for CIA to determine the methods actually used by the child, the data members inherited and the method of the parent that are overridden by the child. Previous IA approaches have state only that a subclass is a child of a changed parent class and that the child should be examined manually for the exact nature of the interactions. CIA does this for the maintainer.

If the class already has a parent, the number of affected methods, objects and descendent classes could range from zero to all. The severity of impact depends on how much of the parent’s structure and functionality is used by the child class. CIA determines the level of dependency by noting which data members of the parent class are referenced in the child and by determining which of the parent’s methods are invoked explicitly.

42

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
class a extends c {
    protected int amem;
    a()
    {
        amem = 0;
    }
}
class c {
    public int cmem;
    public a amem;
    c()
    {
        amem = new a();
    }
}

Figure 4 - 6 Inherits-Contains Interaction

class a extends c {
    protected int amem;
    a()
    {
        amem = 0;
    }
}
class c {
    public int cint;
    protected int cpro;
    private int cpriv;
    void setc(int cv)
    {
        cint = cv;
        a a_in_c =new a();
        a_in_c.cint = 333;
        a_in_c.cpriv = 555;
    }
}

Figure 4 - 7 Parent class using an object of a child

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
4.8.3.1.5 Deletion of an Ancestor Class

Deletion of an ancestor class can cause catastrophic impact on the descendents. Simply removing a class from the system, without further adjustments to the dependent class will cause the code not to compile regardless of the level of dependence between parent and child. All references to public or protected data members or methods will become illegal unless they were overridden in the child. All declarations of objects of this class are now illegal.

4.8.3.1.6 Modification of an Ancestor Class

Modifying an ancestor class can have varying impacts on descendent class depending entirely upon what modifications are made. For example, modifying a method that is subsequently overridden in the descendent has no impact on the descendent. Modifying a method that is not overridden, but is not invoked through the descendent requires recompilation of the descendent class but does not “impact” the descendent.

Modification of a used data member (access attributes only) or modification of a used method may cause some references to these components to become illegal. These changes cause impact in them. The aggregate impact of these types of changes is the impact of modifying an ancestor class. There is no predictable impact that can be determined as caused by “modification of an ancestor class”. Previous research [LI96] has suggested there was. It is assumed in [LI96] that any change to a parent will impact all of its descendents (assuming recompilation is not an impact). As we have stated in section 4.8.3.1.4, the level of impact depends upon the level of interaction between classes and the interactions, if any, that were changed by the maintenance activity. For example, if a parent class has an unused method and the maintainer discovers this
information from the ELLSA, removal of this method engenders no impact on descendents because it is not used.

4.8.3.2 Method Modification Impact Analysis

The impact and ripple effects of modifying a method are caused by changes from: 1) creating or deleting parameters or local variables that are class instances, or 2) deleting or adding calls to other methods.

4.8.3.2.1 Creating, Modifying and Removing Class Instances in Methods

If a method has an instance of a class as a parameter or as a local variable, then the method has an interaction with the class of the instance. This interaction is two-way and is called *uses/used by* interaction. The method is said to *use* the class and the class is said to be *used by* the method. The method can access any public data and/or methods of the class being used. CIA notes the addition of a class instance is as a simple change. There is no impact other than the establishment of an interaction simply by adding the instance, unless the instance is of a class in the hierarchy of this method's class, causing recursive declarations.

In some ways the *uses/used by* interaction is similar to the *contains/contained* relationship between classes, where a class uses more or less of another class. Essentially, the bonds between class and method can be either strengthened or weakened by either adding or deleting interactions. The *uses* interaction can be modified by

- using more or less of the class' methods, thereby altering the call graph
- accessing more or less of the class's data members, thereby increasing or decreasing data dependencies
- deletion of the uses interaction by deletion of the object instance of the class
The *used by* interaction is concurrently modified with the *uses* interaction. Dependencies in the used class will grow or shrink with the modifications to uses method.

### 4.8.3.2.2 Method-to-Method Interaction – Static Methods

In some cases, a *uses* or *used by* relationship exists strictly between methods and not classes. In C++, these types relationships are common, especially in object systems that are not well designed. Strictly speaking, in Java method-to-method interactions always involve a class. All methods in Java belong to a class. Typically, the only manner in which a method can be called without an object instance of its class being declared is if the method is declared as *static*. A static method has at most one copy, and that copy, along with its local data, is maintained for the duration of the program’s execution.

Static method interactions permit the use of “global” data and methods – data and methods that can be used by any method in the system. Modifying static methods or data can impact any class. Because there is no defined relationship or interaction between the static method’s class and the class of a using method, the only means of determining what components are impacted by changes to static methods is by examining the *calls methods* interaction for every method in the system.

In Java, the *uses* relationship as it applies to methods really means "methods that are called by this method". The *used by* relationship as it applies to methods means "methods that call this method" Changes in these relationships of course mean changes in the interactions among the classes and methods. CIA reports the ripple effect of these types of changes.

### 4.8.3.3 Impact Effects from Changing Objects

An object is an instance of a class. The class describes both a structure and behavior for its instances. Methods use object instances. Most of what an object is, its type, its
state, and its identity [KHOS95], is controlled by class structure and method operations. The maintainer can make very few changes to an object instance. Objects can be added or deleted as data members (contained) or as local variables (used by); however, we have already described these actions as changes to a class or method.

An object’s scope can be changed. This change is accomplished by relocating the declaring statement of the object instance. Java and C++ permit declaration statements to occur almost anywhere in a program. The relocation of a declaration may cause references to the object to fall out of scope (causing syntax errors) or it may cause two identifiers with the same name to exist in the same scope.

The most serious impact to the system caused by changing an object arises not from changing the object instance itself but from changing program statements that contribute to an object’s state. This change is done primarily through assignment statements, and to a lesser extent through initialization from parameters (if the instance is in a method). Of course, input operations can alter the state of an object, but there is not a dependency upon other statements or methods via input.

The *assigned from* interactions stored in the ELLSA are the data dependencies that exist for object instances. Adding, deleting, or changing terms in assignment statements (with the object instances as the l-value) alters data dependencies for objects. Altering the object’s data dependencies impacts the set of possible states the object instance might enter. The object ‘s behavior, as well as the value of individual data members is impacted.

The *assigned from* interaction extends to classes, too. The class *assigned from* is the sum of all the *assigned from* interactions of all the object instances of this class. It establishes data dependencies among classes.
4.8.3.4 Summary of Impact Analysis

Maintenance activities on an OO system produce varied amounts of impacts to system components. CIA’s role is to determine the elements that contribute to change of a component and to analyze the effects of change. CIA provides accurate component selection in that it discriminates among components in a class involved in interactions with other classes. Components of client classes are named and their relationship to the changed class is explicitly stated. Earlier IA processes provided generalized results such as “class x uses class y”. CIA provides more refined impact identification than earlier IA processes. It does this by exploiting the architectural clichés present in the OO system, combined with detailed knowledge of the interactions maintained in the ELLSA, to yield component level identification of impacts. CIA provides data on the nature of the component’s relationship with the changed component that helps the maintainer understand why a component is impacted by a change in some other component.

4.8.4 Results of Comparative Impact Analysis

CIA produces a list of system components that have been or will be affected by maintenance activity on the system. The list is called a Comparative Analysis Report (CAR). The affected components are presented along with the interactions they share with other components, if any. The component causing the change is presented also, and if possible, the change or changes that are responsible for the impacts.

4.9 Instrumentation of System Components

One of the main uses of the results of impact analysis is for the preparation of instrumented test versions of the changed and impacted components. The instrumentation process is described in this section. The granularity of instrumentation is at the component level, consistent with the stated granularity of impact analysis.
Classes are not executable components. They cannot be instrumented as a component, but can be instrumented in concept by instrumenting all the methods. Objects themselves are not executable components either. Execution moves an object from one state to another. The state of an object is a collection of values. To track the state of an object requires a runtime environment of great complexity. Creating the runtime environment is beyond the scope of this research.

Only methods execute. Therefore, we instrument methods to record the execution path. The instrumentation instrument is a call to a statically defined method that routes the recorded data to an output file. The contents of the output file are then used to determine which methods were executed. From the trace data CSM can determine the method name and a count of invocations. The names of the method’s invokers are statically obtained from the ELLSA. The execution count is obtained from the trace data.

The trace contains an execution path for the entirety of program execution. The trace records the name of the method and corresponding class.

4.10 Test Coverage Analysis

Test coverage analysis is performed to determine which of the components marked for execution testing were actually executed. CSM uses the ELLSA and the CIA results as a yardstick for comparison against the results of instrumented program execution. If a component has been executed, it is recorded in the program trace. When all components in the list have been recorded in the trace, then every impacted component has been executed at least once.

All the methods that were executed will appear in the trace as a pair of entries: the first entry indicates entry to a method; the second entry indicates a return from a method. Finding these two entries indicates that the method named in the entry is to be marked as
tested in the affected component list. At this point, if a method did not execute, intervention by the maintainer is required to determine why. The data in the trace contains a call graph plotting the execution path, possibly showing why the component did not execute. The maintainer must manually correct the cause.

In many cases, a method that must be executed is involved in an interaction with another component. For example, a method from one class uses a method from another class. A call to the used method does nothing to test the interaction between the methods. Forcing a call of the user method may cause the used method to be called, but there is no guarantee that a control path leading to this call will exist based on the state of the program (unless there is dead code, one should logically exist). In a situation such as this, the maintainer must intervene to insure the provision of the requisite value(s) that cause the call to be made.

In section 4.8.1.1, the simple changes that can be made to a class are presented. The idea is that any and all of these types of changes can cause a class wide impact. The safest course of action in this case is to test everything in the class, which means test all of the methods of the class. However, because it is generally infeasible to retest all methods, the results of CIA help reduce the amount of testing by giving guidance as to which methods need to be tested. It is rarely the case that a change to a class will affect every method, descendent class, container class and uses interaction. In fact, some changes have no impact at all, except for recompilation, and it is assumed that the code being tested by CSM will compile.

The class components tested are obtained by examining the trace created by CSM. The trace names methods executed and their class. A measurement of class testing
coverage is obtained as a simple percentage of class elements (methods) that are exercised in testing.

4.11 Summary

This chapter presented the Comparative Software Maintenance methodology. CSM aids in program understanding by providing the maintainer of OO software a unified model of the interactions and relationships among the class, method and object components of an OO system. CSM provides detailed change impact analysis. CSM creates an instrumented code version of the components affected by maintenance activities on OO systems. CSM then monitors the execution of the affected system to determine testing coverage.

We presented improvements to an existing low-level software architecture model by extending it to represent additional data and new relationships present in OO systems. The ELLSA model contains information on the structure and interactions of components at the level of class, method and object. We presented the formal basis for the ELLSA model.

CSM’s CIA component allows the maintainer of OO software to identify changes in components due to maintenance activities. CIA traces the changes through the system to identify other components that are affected by the changes.
Chapter 5

Predictive Impact Analysis

5.1 PIA Overview

The CSM methodology includes a component called Predictive Impact Analysis (PIA). It allows maintainers to ascertain the impact of proposed changes to a Java system before the changes are committed. PIA uses the ELLSA and comparison algorithms of CIA, as well as the parser and impact analysis algorithms.

In the course of developing CIA and CSM, it became clear that the process of determining affected components could be done by "rule". That is, a certain "simple" change to a system always results in a certain impact. For example, adding a data member always results in no impact except recompilation unless the data member's class was the child of a class possessing a data member by the same name (the new name overrides the old, changing data dependencies, and introducing the possibility of type errors if the data types are not the same.) CSM looks for impacts at the architectural cliché level. Architectural cliché was defined in section 2.4. Changing the cliché in some consistent way should result in a consistent impact. If this is true, then it is possible to predict the impact a change is going to have on a cliché before the change is made.

The ELLSA is a collection of architectural clichés. The clichés are the interactions such as uses or calls, and the relationships such as inheritance and aggregation. The low-level structure of an OO system is a collection of clichéd interactions and relationships. Having knowledge of these clichés in ELLSA, and given that clichés react in a consistent way to a change, it is possible to predict the impact of a maintenance activity or activities before they are performed.
Recall from Chapter 4 the types of changes that can be made to a class. We can add or delete data members or methods, or modify data members or methods. Modeling these types of changes requires a minimum of information about the components. In fact, small changes to the system can be modeled very effectively by the ELLSA model and with a minimum of user intervention. Even the most complex type of "simple" change, the addition of a method, can be modeled in ELLSA with a small amount of user intervention. Some types of changes require only the component's name to be provided by the user. To delete a data member, the user must provide only the data member's name. Maintenance activities of some size can be "performed" as a series of simple changes to the system. Their effects can be gauged individually or as a whole. Section 5.3.3.1 describes all the data inputs required for the various "simple" changes.

Any of the "simple" changes that can be made to a class, a method or an object (except changing an object's scope, which isn't modeled in ELLSA or checked in CIA) can be modeled in PIA. Changes to the "code" are entered in the ELLSA's raw data. The computed portion of the ELLSA is regenerated and the changes are acted upon as if they actually occurred in the code. The changes may be committed, so to speak, by saving them as part of the ELLSA, or they may be abandoned. The ELLSA can be reconstructed at any time in the process. After the changes are made, the ELLSA of the changed "system" can be compared to the original system, just as in CIA, to determine changed components and the impacts created by the changes. We call the changed system a virtual system.

The Predictive Impact Analysis process involves eight steps. Three of the steps are described in Chapter 4. They are noted by an asterisk in the following:

1. creation of the ELLSA for the program structure to be modified*
2. creation of a "virtual" system in which the change will be applied

3. selection of changes to the virtual system

4. application of the changes to the virtual system

5. creation of the ELLSA for the virtual system

6. comparison of the ELLSAs to determine the affected components*

7. ripple effect analysis*

8. results report.

5.2 PIA Process

5.2.1 Creation of the ELLSA

PIA performs IA before the code is actually modified. In order to perform IA, it is necessary to understand the structure and interactions of the code under modification. Acquisition of this understanding is the function of the ELLSA, which provides information about individual internal class structure, external relationships with other classes, the structure of method interfaces, local and parameter objects, call chains, uses and used by relationships, and information about the object instances of an object system. The content and construction of the ELLSA are described in detail in Sections 4.2 and 4.3.

5.2.2 Creation of a Virtual System

The virtual system is the ELLSA of the system after its "modification". The initial state of the virtual system is simply that of a copy of the ELLSA for the original system. The original ELLSA contains the structure of the system prior to changing it. It is not modified in any way. It must be preserved in order to perform change identification.
5.2.3 Selection of Changes to the Virtual System

The maintainer can now begin the process of modifying the virtual system. The changes can be entered in any order and in any number. Changes involving adding new components, or adding to existing components, typically require some detailed information about the change. The knowledge of the system provided in the ELLSA is augmented by the new data being provided.

Changes to the system can be the "simple" changes presented in Section 4.8.1. These changes can all be accomplished with a single dialog between maintainer and PIA. In many cases, however, more than one component must be changed in order to accomplish a given modification. In these cases, new data is needed for all components involved.

The next two sections describe requirements for "simple" changes and what is required for changes that are more complex.

5.2.3.1 Required Input for Simple Changes

The changes that can be applied are repeated below, along with the minimum data input required to complete the change:

- add a data member – requires data type, member name, access attribute. Access defaults to public if no attribute is selected. static, final, and other attributes are optional.
- delete a data member – requires member name only.
- modify access attributes – requires only modified attribute field.
- change a method member – depends on the change. They are
  - add a function call – requires method’s class and name
  - delete a function call – requires method’s class and name
  - add a locally created object instance – requires object’s class and name
• delete a locally created object instance — requires object's name
• delete a method member — requires no data, method is selected from list
• add a method member — depends on the change. They are
  • add parameters — requires data type and name
  • add a function call — requires method's class and name
  • add a locally created object instance — requires object's class and name
• modify object data dependence — requires variable names, objects or function calls

This is an exhaustive list of simple changes of the type that an ELLSA can model. Changes in control path do not effect the ELLSA, except for method calls, so they are not recorded or required.

5.2.3.2 Complex Changes – Changes Involving Two or More Components

Most of the changes described in the previous section would be paired with at least one other simple change to complete a "maintenance activity". For example, if the maintainer adds a method to a class, the logical assumption is that the method will be called by some other method in the system. One or more other methods would be changed to add calls to the method to somehow benefit from its use. Similarly, deleting a used method should be paired with removing the references to it.

5.2.4 Application of Code Changes to the Virtual System

Code changes can be applied in two ways: (1) a single change can be applied to the virtual system and its effect measured singularly; or (2) a set of changes to any or all the existing components can be made and their collective effect measured. The changes are saved as changes to the raw data collected on the original system. The raw data consists of items parsed from the source code such as class name, data members, member
methods, called methods, and object declarations. It is from this "raw" data that the
ELLSA for the virtual system is created.

5.2.5 Creation of the ELLSA for the Virtual System

The ELLSA for the virtual system is created after the user decides to stop the
modifications to the virtual system. The data structures of the ELLSA for the virtual
system contain the same type of information as the ELLSA for the original system. If the
modifications to the virtual system change the structure of the relationships among
classes, methods, and objects, then those changes are recorded in the ELLSA. However,
at this point they are not recognized as changes. That knowledge will be available to PIA
after the ELLSA of the original system and the ELLSA of the virtual system are
compared.

The ELLSA can be regenerated after each change. The regeneration of the ELLSA
can help the maintainer determine if other changes are required based upon a previous
change. For example, assume the maintainer is removing a method call from a method.
Upon regenerating the ELLSA, the called method’s ELLSA indicates that it no longer
called by any methods. It could now be considered to be dead code and removed from the
system.

The ELLSA can be regenerated after all the virtual maintenance activities have been
performed. This approach is similar to CIA, in that CIA would not normally be
performed until after the system has been modified completely. When PIA has all the
proposed changes, the maintainer receives a complete description of the effect on the
system as a whole.

ELLSA comparison was described in Section 4.7.2. PIA uses the same routines as
CIA for comparison.
REA for PIA is performed the same as in CIA. It is presented in section 4.8.3.

5.2.6 Presentation of the Results

PIA produces a list of potentially affected components called a Comparative Analysis Report (CAR). In addition to the component name, its location is noted. For class components, descendant and ancestor classes (if any) of the affected class are noted. Contains and container class relationships, as well as object family and uses relationships are noted.

For affected method components, the report shows the name and location of the method, its call graph, and object components. Object component data that are reported includes the name, location, and static type of the object.

Figure 5 – 1 shows a PIA report on the effects of modifying data dependencies between objects of two different classes. Some terms in assignment statements involving the objects were deleted, and some were added. The result of these changes affects the objects and the declaring method. The CAR reports the files involved in the original system. Each changed or affected component is listed. Any interactions maintained by the component is displayed.

5.3 Summary

PIA is used to predict the effect of changes to a system before they are made. PIA is useful for determining resource allocation and testing requirements of maintenance activities to be performed on a system.
This file contains a listing of affected components from project postobjdd.jpj.
postobjdd.jpj contains the following files:
C:\V\CSM\ModifyObjDD\ObjectDDMOD\Class1.java

postobjdd.jpj was compared to preobjdd.jpj and the following changes and impacts were discovered:

METHOD:Class1.main ALTERED
Location:C:\V\CSM\ModifyObjDD\Class1.java
Deleted method call(s):
   svar.getprivar
Added method call(s):
   Math.sin
   Math.max

OBJECT:newsvar ADDED in postobjdd.jpj
Location:C:\V\CSM\ModifyObjDD\ObjectDDMOD\Class1.java
Declared in:main

OBJECT:dvar ALTERED
Location:C:\V\CSM\ModifyObjDD\Class1.java
Deleted data dependencies:
   svar.getprivar
Added data dependencies:
   newsvar.pubvar
   Math.sin

OBJECT:svar ALTERED
Location:C:\V\CSM\ModifyObjDD\Class1.java
Added data dependencies:
   Math.max

---

Figure 5 - 1 Affected components
Chapter 6

JFlex – a Tool for Performing CSM

6.1 Introduction

CSM provides the maintainer of Java software systems with several valuable functions. The ELLSA models the software, CIA and PIA determine changed components and determine or predict, respectively, the impact of changes to the systems, and CSM’s testing assistance helps insure all the modified or impacted components are executed at least once. Large quantities of data about the system are generated in order to perform CSM. These data and functions can be manually computed. However, the time and labor required to do so is prohibitive. One of the goals of this research is to ease the maintenance process by providing automated support for program comprehension, impact analysis and test monitoring. In doing so, we address the tool support problem that is presented in Chapter 1. Automation of the CSM process is embodied in JFlex.

JFlex is written in C++ using Microsoft Foundation Classes in Visual C++. JFlex runs in the Microsoft Windows 95/98/NT environments. It is composed of 25 classes and, with the exception of the parsing functions, adheres to OO design principles throughout. (The parser was generated by lex/yacc software as procedural code.) The organization of JFlex is shown in Figure 6-1.

JFlex conceptually has four main “modules”. Each module performs a major task of the software. The modules have these functions:

1. parsing function

2. ELLSA construction functions that discover the interactions and relationships in the original source code and the viewing functions
3. ELLSA comparison, impact analysis and reporting functions

4. code instrumentation, execution and testing analysis functions

---

Figure 6 - 1 JFlex data structures for parsing and ELLSA construction / viewing

These "modules" are spread over several classes and many methods. The modules depend upon each other in series; thus their actions must be carried out in order. The modules are presented in the sections 6.3, 6.4, 6.5 and 6.6. In section 6.2, the data structures used in JFlex are described.

**6.2 JFlex Data Structures**

The ELLSA of a software system is a collection of rooted component subgraphs, each subgraph being represented by a textual description, or TD. The TDs are classes in JFlex.
The three components — classes, methods and objects — are each represented in a class. The common elements of each class (symbol name, file location, attributes) are defined in a common ancestor class called CCMO (Common to Classes, Methods and Objects). Each component class inherits that part of its structure from CCMO. JFlex employs a separate class for each type of component. Each class is a contained class within the ELLSA document class. Figure 6-2 shows the ELLSA data structures class structure in JFlex.

Some data are common to class, method, and object and appear to be stored in all three structures. There is no duplication, however. Pointers in the form of array indices are employed throughout JFlex to allow access to members of other components.

In the ELLSA view it appears as if there are two sets of data tables, raw data and ELLSA data. In fact there is only one set of tables. The “raw” data is the data extracted from the source code. Much of the ELLSA is “raw” data that is directly extractable from the source code of the system. These data include class names, method names, file locations, and class data members. All the computed data (i.e. static interface, contains, etc) of the ELLSA are added when the ELLSA algorithms are invoked.

6.3 JFlex Parsing Functions

The first function of CSM is to reverse engineer the software system in order to establish its architectural structure. The process of reverse engineering begins with determining the files contained in the software system. This process in itself requires extensive knowledge of the system. While CSM does not consider file names or locations (except as static display data) in computing the ELLSA, it must be given the names of the files containing the system. This data is stored in the document class, and is
saved to disk as part of the "raw" data. The maintainer may edit the file list. Files may be added or removed.

![Diagram of ELLSA Internal Class Structure]

The goal of the parsing module is to create a set of tables containing the "raw" data of the software system. The tables are initially empty (nothing is assumed about their content) and are statically sized, but may be dynamically enlarged is necessary by the `realloc` function in C++. These tables are used temporarily while the parser operates. Upon completion of the parsing functions, the data are copied into the "document" class in JFlex, and the parsing tables are deleted.

**6.3.1 Token Scanner and Parser**

The scanner and parser in JFlex were automatically generated by Parser Generator [STE98], a lex and yacc work-a-like shareware program for Windows. It generates C code functions and tables for creating LR(1) parsers. The BNF grammar from which the scanner and parser are generated is from [BRO98]. It is compliant with Java 1.1 and supports inner classes and anonymous classes. The scanner recognizes comments. It was correct except for recognizing floating-point constants, a problem easily corrected.
Parsing of source code is a two-step process. First, a lexical analyzer, or scanner extracts tokens. Tokens are reserved words, identifiers and special characters (operators such as *, ->, +, etc). Second, the tokens are used as input to an algorithm that is constructed from a set of rules that describe valid sequences of tokens. The sequences are statements and declarations of a programming language.

The parser's job in JFlex is to locate the components of interest to CSM – classes, methods and objects. The parser can do this job because it recognizes constructions of the Java language. As the parser scans and recognizes the constructs of Java, it halts while a JFlex function that is called by the actions that are embedded in the grammar extracts data about a component. After JFlex has secured its data, the parser continues.

Class names, data members, method declarations and body are all directly available in Java from within the class declaration. Object components, however, cannot be extracted directly from the code but must be "computed". After all code is parsed, data types of method parameters and local variables are checked to determine if they are objects. Objects, in Java, may be declared prior to the declaration of their class. The parser assumes that it will encounter the declaration of the class at some point, so it does not issue an error message or warning. After parsing, these anonymous declarations must be resolved.

6.4 ELLSA Construction and Viewing Functions

Some of the content of the TDs for the system components is available directly from the source code. This includes the component name, file name, class affiliation, parameter, local variables, immediate inheritance (upwards), and implements associations. The remainder of the contents must be created or discovered by algorithm. The algorithms used are based on [SHR96] and some have been adjusted to meet the
specifics of Java. For example, the static interface of a class is almost identical in C++ and Java; however, differences in the manner that methods are overridden in Java necessitated changes to the algorithm that computes the static interface. On the other hand, there are no dynamic interactions, as defined in [SHR96], that exist in Java. These algorithms are not required. The Static Data Interface and Assigned By interactions for objects and classes are new in ELLSA. Algorithms to compute these interactions were developed for this research. The Assigned By algorithms are presented in sections 6.4.1.1 through 6.4.1.3.

6.4.1 Constructing ELLSA Architectural Clichés

The architectural clichés of an OO system are obtainable essentially by inspection. Doing so in a complex or large system, however, is error prone, tedious and time consuming. With the help of automation, the architectural clichés of an OO system are quickly obtainable.

6.4.1.1 Computing Trivial Architectural Clichés

The computational effort involved in finding many of the architectural clichés is expended primarily in looping through the three component tables, comparing component names, and assigning a link between two components in the appropriate interaction list. Most of the interactions of the ELLSA are obtained in this manner.

Algorithms to compute the following interactions and relationships are trivial:

1. object family
2. descendent classes
3. ancestor classes
4. container classes
5. contained by
6. uses methods
7. used by methods
8. methods called by member method of the class

Implementation of these algorithms typically consists of a nested for loop structure, selection statements, and assignment to a list.

6.4.1.2 Computing Complex Interactions

The non-trivial relationships and interactions have more complex algorithms. [SHR96] provides the implementation algorithms. Some of the interactions were applicable without change to Java and some required modification. They are:

- Static Interface of a class – this algorithm had to be refined to accommodate differences in inheritance between Java and C++. Specifically, Java overrides inherited methods only if the method signature matches exactly (excluding return type). C++ overrides all methods of a parent class that have the same name as a method of the subclass.

- Dynamic Interface of a class – computed without change

- Dynamic Interface of a function (method) – computer without change

- Dynamic Interface of an Object – this algorithm was modified to accommodate differences between Java and C++. Specifically, Java does not use pointer objects. Objects are references in Java. The dynamic interface indicates the data type that an object may assume as a parameter passed into a function.

6.4.1.3 Static Data Interface of a Class

The Static Data Interface of a class is a new relationship in CSM. It is the union of the non-private data members of the class' ancestor classes and all of its data member.
members from ancestor classes that are overridden in subclasses are excluded from the static data interface. If a class has no ancestor or all data members in its ancestors are private, the static data interface of the class is its data members. The static data interface list is computed by traversal up the inheritance tree of a class component. Java does not support multiple inheritance, so a class will have at most one immediate ancestor. Finding the next ancestor is accomplished by inspection of the current ancestor's parent_class attribute in its class TD. Figure 6-3 presents the algorithm for determining the static data interface of a class.

6.4.1.4 Assigned From Interactions for Classes and Objects

The assigned from interactions on objects and classes are presented. The interactions are a set union of the existing dependencies in the source code for an object, and through objects, for classes. All data members of the object participate in this interaction. All objects of the class contribute to the assigned from dependency for a class. The algorithm for the assigned_from interaction for an object O is presented in Figure 6-4. The assigned_from interaction for a class is the collection of assigned_from interactions belonging to its objects. The algorithm is presented in Figure 6-5.

Some non-object sources of terms in an assignment expression are recorded in the assigned_from interaction. These sources include function calls. Data sources such as constants and variables that are not members of a class are not recorded. The data dependencies measured are those among objects. All methods belong to some class.

6.4.2 JFlex Viewing Functions

The standard Windows document/view model is employment in JFlex. In this model the data structures and operations on the data – the document, are separated from visualizations of the data – the view. This very powerful model allows the document data
to be presented in a variety of styles and in segments. The "document" in the case of JFlex is the raw data, the ELLSA model, the IA data, and the test coverage data. The "views" are the three output areas presented in JFlex - the ELLSA structures tree, code window and the system/component information tree. Figure 6-6 shows the full JFlex screen. The three panes shown are always present. Inputs, reports and results are shown in output dialog boxes.

Let C be the class, SD the static data interface, DM its data members, and CAC its current ancestor class.

Input: C, DM, CAC
Output: SD

Algorithm:

1. For each class
2. SD = C.DM //its own data members
3. CAC = ancestor of C
4. While CAC exists
   (a) For all CAC.DM
      (i) If CAC.DM is non-private AND CAC.DM \in SD then
      (ii) SD = SD \cup CAC.DM
   (b) CAC = ancestor of CAC

Figure 6-3 Algorithm for Static Data Interface

\[ \text{O.assigned\_from} = \text{NULL} \]
For all assignment statements with O as the l-value
\[ \text{O.assigned\_from} = \text{O.assigned\_from} \cup \text{r-value} \]
where r-value is an expression involving qualified data members or functions of any class, including O's, or local variable or parameter in a method, and l-value is the left hand side of an assignment operator

Figure 6-4 Assigned From Interaction for Objects Algorithm

\[ \text{C.assigned\_from} = \text{NULL} \]
For all objects of family class C
If(C.O.assigned\_from is a class)
\[ \text{C.assigned\_from} \cup \text{C.O. assigned\_from} \]

Figure 6-5 Assigned From Interaction for Classes Algorithm

68

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
6.5 ELLSA Comparator

The ELLSA comparator has the function of determining what changes have occurred to a Java system as a result of maintenance activities. The types of changes that can be performed are described in section 4.8.1. The comparator scans the ELLSA of the original system and compares it to the ELLSA of the modified system, component by component. As changes are found the component is marked for impact analysis and the nature of the change is recorded. Classes are checked first, then methods, and then objects.

6.5.1 Class Change Identification

Class change identification algorithms determine if and what changes were made to a class. A change to a method is not considered a change to a class until it is demonstrated that the change somehow affects the class.
The comparator first determines if any classes have been added or deleted. The comparator then compares classes with the same name from the pre- and post-modified systems. The comparison of two classes is done in two steps finding changes in data members first, and then changes to method interfaces.

6.5.2 Method Change Identification

Recall from section 4.8.1.2 that a change to a member method involves one or more of the following actions:

- add a function call
- delete a function call
- add a locally created object instance or variable
- delete a locally created object instance or variable

There are many more ways to change a method, but from the point of view of the ELLSA, these other changes are irrelevant. Note that changes in method signature constitute a change to the defining class of the method. This type of change is considered to be deleting a method and adding a new method.

The process of comparing methods is complicated by name polymorphism. Because of name polymorphism on methods in Java, the comparator must match methods on name and parameter data types. The algorithm attempts to match a method from the modified system with a method from the original system first by name, class, and number of parameters, and then order and type of parameters. If a matching signature is found, the comparator then compares internal details of the method, looking for the four types of changes listed at the start of this section.
6.6 Comparative Impact Analysis and Predictive Impact Analysis

6.6.1 CIA in JFlex

CIA begins after change identification is complete. Changed components have been noted, along with the changes that have been made to them. CIA uses the knowledge of the system structure from the ELLSA, along with knowledge of the changes made to a component, to perform impact analysis. A series of short algorithms are applied to the changed system components to trace impacts and produce a list of affected components.

The impacts described in section 4.8.3 are traced through the relationships and interactions of the changed components. The algorithms that implement impact analysis consist mainly of short groups of selection statements in loop structures and are, for the most part, straightforward.

The results of change identification and impact analysis are presented in a dialog box. Changed components are displayed and the impacts, if any, of the changes are presented. Incidental information, such as subclasses, container classes and the object family of a class, are noted as well even if they are not affected by the change.

6.6.2 PIA in JFlex

PIA can be performed with JFlex. The initial steps are the same as CIA: parsing the code, constructing the ELLSA, creating the viewing structures. Selecting a menu command in JFlex then creates the virtual system. The raw data and interactions of the original system are replicated in the virtual system. The system is then compiled to create objects and to create the tree viewing structures in JFlex.

All changes are applied to components of the virtual system through a dialog box. In the dialog, the maintainer provides data about the change. If a change is being made to a class, the class name must be selected (highlighted) in JFlex. If the change is being made...
to a method, the method name must be highlighted, as with objects. The type of change required is selected from a list of changes and is automatically applied to the highlighted component after the dialog is completed, providing the dialog terminated normally.

Upon completion of the modifications the maintainer reconstructs the ELLSA for the virtual system. Impact analysis can now occur. The impact analysis process for PIA uses the same functions as CIA. The results are presented in a CAR.

6.7 Code Instrumentation, Compilation and Execution

The instrumented code is created by a mini-parser developed for this research. The parser recognizes the start and end of a method's executable statements. Each of these actions is instrumented with calls to a method that records data appropriate to the action. These method calls create data about the execution path. This data is written to disk as the Java program executes.

The instrumented java code is compiled through a system call that launches a DOS batch file. The instrumented, compiled system is executed through a menu command that invokes a system command that invokes the java interpreter that runs the class containing the function "main". As the program executes, the instrumented statements produce output that is written to disk. Figure 6 - 7 shows an example trace and the source code.

The instrumenting statement begins with "fileout.writeit". This method is defined in the fileout class, along with the output stream initialization. The trace entries are marked with a "+" for entering a method and a "-" for exiting a method. The instrumentation is completed with the method's name. All return statements are instrumented. The instrument statement and the return are placed within a bracket pair in all cases to group the two in the case of the return statement following a control structure. If a method has only a return statement, the two instrumenting statements are contiguous. If the last
```java
+Class1.main
+a.a
-a.a
+b.b
-b.b
+a.bye
-a.bye
-Class1.main

public class Class1
{
    public static void main (String[] args)
    {
        fileout.initializeFile();
        fileout.writeit("+Class1.main");
        a avar = new a();
        b bv = new b(7);
        avar.bye();
        fileout.writeit("-Class1.main");
    }
}

public class a
{
    public a()
    {
        fileout.writeit("+a.a");
        fileout.writeit("-a.a");
        return ;
    }

    public void hi()
    {
        fileout.writeit("+a.hi");
        fileout.writeit("-a.hi");
        return ;
    }

    public void bye()
    {
        fileout.writeit("+a.bye");
        System.out.println("Hello from bye");
        fileout.writeit("-a.bye");
    }
}

public class b
{
    public int y;
    public b(int avar)
    {
        fileout.writeit("+b.b");
        y = avar;
        fileout.writeit("-b.b");
    }
}

Figure 6–7 Execution trace and source code
```

73

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
physical statement in the method is not a return, then an instrument statement is placed as the last statement, marking the exit point of the code.

6.8 Test Coverage Analysis

Test coverage analysis is performed after execution of the Java system under examination. It is a straightforward operation involving the trace data produced during execution of the system and the affected components list. The two are compared to determine if the affected method components have been executed and the percentage of impacted method components executed. JFlex presents the results in a dialog. Figure 6 – 8 presents the output dialog produced by JFlex when a request for test coverage is made. The first column is the execution trace with the entries as described above. The second column is the list of affected methods that were marked for retesting. The third column lists the methods that were found in the trace that were also marked for retesting; in other words, the affected methods that were actually executed. On the far right, a text box presents the percentage of coverage for methods on this particular run.

6.9 Summary

JFlex demonstrates that the CSM process can be automated. Other than providing the names of the system’s files, the maintainer has only to activate menu commands to perform CSM on a modified Java System. The original source is not required as long as the ELLSA of the original system has been preserved. In the next chapter we presented examples by which JFlex was validated. We also present a case study validating the CSM process.
Figure 6–8 Measuring the test coverage
Chapter 7

Validation of the CSM and JFlex

7.1 Introduction

We validated this research in the following ways. First, we show that CSM is a feasible methodology. This is demonstrated by its implementation in JFlex. Second, we show that JFlex gives correct results, demonstrated by comparing results of the processes with manually computed results. Third, we show that CSM improves the maintenance process. This is demonstrated by results produced by CSM that provide change and impact information and testing support for the maintenance phase.

7.2 Demonstrating the Feasibility of CSM, Correctness of ELLSA and JFlex

JFlex implements the algorithms required to create the software architecture of a Java system, perform impact analysis, and generate test suites.

The required relationships for creating the ELLSA of the system, are rendered as part of the output of JFlex. The results of the process implemented in JFlex are identical to ELLSA's created manually. In some cases, the JFlex found relationships that were missed by the manual preparer, but that were rightly included.

We tested CSM and JFlex under multiple "program scenarios". Program scenarios are complete Java programs which contain at least one, and in many cases more, commonly occurring OO architectural clichés. The scenarios range from single class programs to systems containing 6 classes, 26 methods and 5 objects. The scenarios were coded as Java software systems and compiled and executed to confirm their syntactical correctness. The scenarios include:

1. Primitive variable declaration
2. Single class Java program with no interactions other than system calls
3. Complex method interactions
4. Object data dependencies
5. Two class Java system with inheritance
6. Two class Java system with a contains/contained relationship
7. Two class Java system with uses/used relationship
8. Three class Java system with inheritance and contains/contained relationship
9. Three class Java system with inheritance and uses/used relationship
10. Four or more class Java system with inheritance, containment, and uses

To conserve space, we present only the source code and ELLSA for scenarios 2, 5 and 10.

7.2.1 Scenario 2 – Single Class Java System

In scenario 2, JFlex analyzes a single class Java system. The class has one data member and two methods. This example demonstrates the ability of JFlex to statically comprehend the structure of a class. Table 7 - 1 presents the source code for this scenario. The code defines a single class with no interactions other than “system” calls – calls to methods that are defined as components in packages delivered with Java. These “system” methods are similar to libraries in C++.

Table 7 - 2 presents the ELLSA for this scenario. All class components required are presented – data members, methods, and interactions are in the ELLSA. The only interactions present are between the two member methods of the class. Main calls stringcopy. That event is recorded for both methods. The other methods called by main are system library components and are not part of the original system. These are recorded to presented the maintainer with complete call path information. It is highly unlikely that the maintainer will need to modify system calls.
public class StringDemo
{
    static String copy;
    
    public static void main(String[] args)
    {
        String sentence = "Text processing is hard!";
        int position;
        
        position = sentence.indexOf("hard");
        System.out.println(sentence);
        System.out.println("012345678901234567890123");
        System.out.println("The word \"hard\" starts at index "+ position);
        
        sentence = sentence.substring(0, position) + "easy!";
        System.out.println("The changed string is:");
        System.out.println(sentence);
        stringcopy(sentence);
        
        System.out.println("Press enter key to end program.");
    }
    
    static void stringcopy(String str)
    {
        copy = str;
    }
}
Class: StringDemo
Location: C:\V\ELLSASingleClass\new\StringDemo.java

Data Members
- copy : String : Static

Member Methods
- StringDemo.main(String args)
- StringDemo.stringcopy(String str)

Static Interface
- StringDemo.main(String args)
- StringDemo.stringcopy(String str)

Calls functions
- sentence.indexOf
- System.out.println
- sentence.substring
- stringcopy

METHOD: StringDemo.main(String args)
File: C:\V\ELLSASingleClass\new\StringDemo.java
Local variables
- String sentence
- int position
Calls functions
- sentence.indexOf
- System.out.println
- sentence.substring
- stringcopy

METHOD: StringDemo.stringcopy(String str)
File: C:\V\ELLSASingleClass\new\StringDemo.java
Called by Methods:
- main
7.2.2 Scenario 5 – Two Class Java System with Inheritance

Java supports class inheritance as explained in Section 2.6. Subclasses do not inherit private data members and methods. Protected and public members are inherited by subclasses. If the inheritance passes over multiple levels, all of the public and protected data members and methods are inherited, excepting, of course, those that are overridden by declaration in lower levels. CSM discovers and records the relationships between parent and child classes over any number of levels.

There are three levels of inheritance in this scenario. Some methods and data members are overridden by the subclasses. To test JFlex closely, some method names are very close to names in subclasses – different by only the case on one letter. Method getMissles() is declared in class aircraft and low_level, but class bomber, sandwiched between those two classes, has a method called getmissles(). Some method and data are private in parent classes and cannot be inherited. Scenario 5 demonstrates this situation also. Table 7 – 3 gives the source code for the three classes in the inheritance relationship. Table 7 – 4 shows the ELLSA interactions and relationships.

Note that private member stealth and method getStealth are not inherited in subclasses, and are not in the static interface of bomber or low_level. The private members of bomber are likewise unavailable to low_level. Low_level does inherit the protected and public members. Low_level does not inherit getMissles from class aircraft. It does inherit getmissles from bomber. The parsing and ELLSA functions correctly maintain case sensitively and carefully discriminate between very similar names and declarations. CSM constructs the correct static interface through two levels of inheritance. This scenario demonstrates the ability of CSM and JFlex to correctly trace and model multiple levels of inheritance.
public class aircraft
{
    protected int Crewcount;
    protected int Numeng; //number of engines
    protected int Propulsion; //
    protected int gtw; //gross takeoff weight
    private int stealth; // stealth rating not inherited
    aircraft(int cc, int ne, int g, int p)
    {
        Crewcount = cc;
        Numeng = ne;
        gtw = g;
        Propulsion = p;
    }
    aircraft()
    {
    }
    public int getCrewcount()
    {
        return Crewcount;
    }
    public int getNumeng()
    {
        return Numeng;
    }
    public int getGtw()
    {
        return gtw;
    }
    public int getPropulsion()
    {
        return Propulsion;
    }
    private int getStealth()
    {
        return stealth;
    }
    public int getMissles() //overriden by low_level but not bomber
    {
        return 0;
    }
}
public class bomber extends aircraft
{
    private int Bombload;
    private boolean Nuclear;
    protected boolean infrared;
    protected int missles;
    bomber(int cc, int ne, int gtw, int p)
    {
        super(cc, ne, gtw, p);
    }
    bomber() {}
public void setBombload(int bl) {
    Bombload = bl;
}

public void setNuclear(boolean n) {
    Nuclear = n;
}

public int getMissles() {
    return missles;
}

public class low_level extends bomber {
    private int missles; // overrides missles from bomber
    private int chaff;
    private boolean garadar;
    low_level() {
        super();
    }
    public int getMissles() {
        return missles;
    }
}
Class: low_level  
Location: C:\Wellsainherits\low_level.java

Data Members
- misses : int : Private //member overrides bomber’s misses
- chaff : int : Private
- garadar : boolean : Private

Inherited Members //inherits from two levels
- bomber.infrared
- aircraft.Crewcount
- aircraft.Numeng
- aircraft.Propulsion
- aircraft.gtw

Member Methods
- low_level.low_level()
- low_level.getMissles()

Ancestor Classes
- aircraft
- bomber

Static Interface
- low_level.low_level()
- low_level.getMissles() //overrides aircraft’s getMissles()
- bomber.bomber(int cc,int ne,int gtw,int p)
- bomber.bomber()
- bomber.setBombload(int bl)
- bomber.setNuclear(boolean n)
- bomber.getmissles() //but not bomber’s getmissles()
- aircraft.aircraft(int cc,int ne,int g,int p)
- aircraft.aircraft()
- aircraft.getCreacount()
- aircraft.getNumeng()
- aircraft.getGtw()
- aircraft.getPropulsion()

Calls functions
- super

-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Class: bomber  
Location: C:\Wellsainherits\bomber.java

Data Members
- Bombload : int : Private
- Nuclear : boolean : Private
- infrared : boolean : Protected
- misses : int : Protected

Inherited Members
- aircraft.Crewcount
- aircraft.Numeng
- aircraft.Propulsion
- aircraft.gtw

Member Methods
- bomber.bomber(int cc,int ne,int gtw,int p)
- bomber.bomber()
- bomber.setBombload(int bl)
- bomber.setNuclear(boolean n)
- bomber.getmissles()

Ancestor Classes
- Aircraft
Descendent Classes
  low_level

Static Interface
  bomber.bomber(int cc, int ne, int gtw, int p)
  bomber.bomber()
  bomber.setBombload(int bl)
  bomber.setNuclear(boolean n)
  bomber.getmissles()  // bomber has both getmissles()
  aircraft.aircraft(int cc, int ne, int g, int p)
  aircraft.aircraft()
  aircraft.getCrewcount()
  aircraft.getNumeng()
  aircraft.getGtw()
  aircraft.getPropulsion()
  aircraft.getMissles()

Object Family
  bl in Method:Class1:main File:C:\V\ellsainherits\Class1.java

Created by class
  Class1

Calls functions
  super

-------------------------------

Class:Class1
Location:C:\V\ellsainherits\Class1.java

Data Members
  Class1.main(String args)

Static Interface
  Class1.main(String args)

Creates objects of class
  bomber

-------------------------------

Class:aircraft
Location:C:\V\ellsainherits\aircraft.java

Data Members
  Crewcount : int : Protected
  Numeng : int : Protected
  Propulsion : int : Protected
  gtw : int : Protected
  stealth : int : Private

Member Methods
  aircraft.aircraft(int cc, int ne, int g, int p)
  aircraft.aircraft()
  aircraft.getCrewcount()
  aircraft.getNumeng()
  aircraft.getGtw()
  aircraft.getPropulsion()
  aircraft.getStealth()
  aircraft.getMissles()

Descendent Classes
  bomber
  low_level

Static Interface
  aircraft.aircraft(int cc, int ne, int g, int p)
  aircraft.aircraft()
METHOD: low_level.low_level()
File: C:\V\ellsainherits\low_level.java
Calls functions
super

METHOD: bomber.bomber(int cc, int ne, int gtw, int p)
File: C:\V\ellsainherits\bomber.java
Calls functions
super

METHOD: Class1.main(String args)
File: C:\V\ellsainherits\Class1.java
Local variables
bomber bl
Creates Objects of Class
bomber

METHOD: aircraft.aircraft(int cc, int ne, int g, int p)
File: C:\V\ellsainherits\aircraft.java

METHOD: bomber.bomber()
File: C:\V\ellsainherits\bomber.java

METHOD: bomber.setBombload(int bl)
File: C:\V\ellsainherits\bomber.java

METHOD: bomber.setNuclear(boolean n)
File: C:\V\ellsainherits\bomber.java

METHOD: bomber.getmissles()
File: C:\V\ellsainherits\bomber.java

METHOD: aircraft.aircraft(int cc, int ne, int g, int p)
File: C:\V\ellsainherits\aircraft.java

METHOD: aircraft.aircraft()
File: C:\V\ellsainherits\aircraft.java

METHOD: aircraft.getCrewcount()
File: C:\V\ellsainherits\aircraft.java

METHOD: aircraft.getNumeng()
File: C:\V\ellsainherits\aircraft.java

METHOD: aircraft.getGtw()
File: C:\V\ellsainherits\aircraft.java

METHOD: aircraft.getPropulsion()
File: C:\V\ellsainherits\aircraft.java
<table>
<thead>
<tr>
<th>File: C:\V\ellsainherits\aircraft.java</th>
</tr>
</thead>
<tbody>
<tr>
<td>METHOD: aircraft.getStealth()</td>
</tr>
<tr>
<td>File: C:\V\ellsainherits\aircraft.java</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>METHOD: aircraft.getMissles()</td>
</tr>
<tr>
<td>File: C:\V\ellsainherits\aircraft.java</td>
</tr>
<tr>
<td>--------------------------------------</td>
</tr>
<tr>
<td>OBJECT: bomber bl</td>
</tr>
<tr>
<td>File: C:\V\ellsainherits\Class1.java</td>
</tr>
</tbody>
</table>

**Dynamic type(s)**
- low_level

**Static Interface**
- bomber: Constructor
- bomber: Constructor
- setBombload: void
- setNuclear: void
- getMissles: int

**Actual Interface**
7.2.3 Scenario 10 – Java System with Inheritance, Containment, and Uses

Most complex OO systems possess many OO architectural clichés. In Scenario 10 presents the same Java system used in Scenario 9 but with the addition of a contains relationship in the highest level ancestor class, Employee. The contained class, EmpInfo, is a simple data repository for employee information with a method to set the data. An instance of EmpInfo is a data member in Employee. Calls to the data setting method are made from class Test.main(). All other relationships remain unchanged.

JFlex correctly located the uses/used, contains/contained by and inheritance relations. JFlex correctly located the JFlex method calls from with the container class Employee to the contained class EmpInfo. The inherited data member empAddress was located and recorded. The Actual Interface of the objects are shown correctly in the ELLSA. Table 7 – 5 presents a very complex, but abbreviated ELLSA. The Static Interface of class and object components have been removed.

In this more complex example, class Test declares objects of five different classes. The object ref of class employee provides a polymorphic interface for the other four objects. At various times in execution, ref is an object of CommissionWorker, HourlyWorker, PieceWorker and Boss. The TD notes this situation for object ref as the dynamic types of ref. The data dependencies for ref are clearly defined in its TD. The actual interface for ref appears to contain redundant information. Its contents are repeated because each object identity that ref assumes contains methods of the same name. CSM presents this instance of dynamic binding of a set of objects through the dynamic interface of the object and the object’s data dependencies. The method calls listed in method main confirm the presence of this form of polymorphism.
Table 7-5 ELLSA for Scenario 10

Class: Test
Location: C:\V\ellsa_I_C_U\Test.java

Data Members

Member Methods

Test.main(String args)

Creates objects of class

Employee

Boss

CommissionWorker

PieceWorker

HourlyWorker

Calls functions

Boss

b.empAddress.setAddress

CommissionWorker

c.empAddress.setAddress

PieceWorker

p.empAddress.setAddress

HourlyWorker

h.empAddress.setAddress

DecimalFormat

ref.toString

precision2.format

ref.earnings

b.toString

b.earnings

c.toString

c.earnings

p.toString

p.earnings

h.toString

h.earnings

JOptionPane.showMessageDialog

System.exit

Class: CommissionWorker
Location: C:\V\ellsa_I_C_U\CommissionWorker.java

Data Members

salary : double : Private

commission : double : Private

quantity : int : Private

Inherited Members

Employee.empAddress

Member Methods

CommissionWorker.CommissionWorker(String first, String last, double s, double c, int q)

CommissionWorker.setSalary(double s)

CommissionWorker.setCommission(double c)

CommissionWorker.setQuantity(int q)

CommissionWorker.earnings()

CommissionWorker.toString()

Ancestor Classes

Employee

Object Family

c in Method: Test.main File: C:\V\ellsa_I_C_U\Test.java

88

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Created by class Test
Calls functions super setSalary setCommission setQuantity

Class: EmpInfo
Location: C:\V\ellsa_I_C_U\EmpInfo.java
Data Members
  Address1: String: Private
  City: String: Private
  State: String: Private
  Zip: String: Private
Member Methods
  EmpInfo.EmpInfo()
  EmpInfo.setAddress(String A1, String C, String S, String Z)

Container classes
  Employee

Class: Employee
Location: C:\V\ellsa_I_C_U\Employee.java
Data Members
  firstName: String: Private
  lastName: String: Private
  empAddress: EmpInfo: Protected
Member Methods
  Employee.Employee(String first, String last)
  Employee.getFirstName()
  Employee.getLastName()
  Employee.toString()
  Employee.earnings()

Descendent Classes
  Boss
  PieceWorker
  HourlyWorker
  CommissionWorker

Object Family
  ref in Method: Test.main File: C:\V\ellsa_I_C_U\Test.java
Contains classes
  EmpInfo
Created by class Test
Calls functions EmpInfo

Class: HourlyWorker
Location: C:\V\ellsa_I_C_U\HourlyWorker.java
Data Members
  wage: double: Private
  hours: double: Private
Inherited Members
  Employee.empAddress
Member Methods
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HourlyWorker.HourlyWorker()</td>
<td>String first, String last, double w, double h</td>
<td>Initialize HourlyWorker with first name, last name, wage rate, and hours worked.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set wage rate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Set hours worked.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Calculate earnings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Return as a string.</td>
</tr>
</tbody>
</table>

Ancestor Classes
- Employee

Object Family
- h in Method: Test.main File: C:\ellsa_I_C_U\Test.java

Created by class Test

Calls functions
- super
- setWage
- setHours

Class: PieceWorker
Location: C:\ellsa_I_C_U\PieceWorker.java

Data Members
- wagePerPiece : double : Private
- quantity : int : Private

Inherited Members
- Employee.empAddress

Member Methods
- PieceWorker.Pieceworker(String first, String last, double w, int q)
- PieceWorker.setWage(double w)
- PieceWorker.setQuantity(int q)
- PieceWorker.earnings()
- PieceWorker.toString()

Ancestor Classes
- Employee

Object Family
- p in Method: Test.main File: C:\ellsa_I_C_U\Test.java

Created by class Test

Calls functions
- super
- setWage
- setQuantity

Class: Boss
Location: C:\ellsa_I_C_U\Boss.java

Data Members
- weeklySalary : double : Private

Inherited Members
- Employee.empAddress

Member Methods
- Boss.Boss(String first, String last, double s)
- Boss.setWeeklySalary(double s)
- Boss.earnings()
- Boss.toString()

Ancestor Classes
- Employee

Object Family
- b in Method: Test.main File: C:\ellsa_I_C_U\Test.java

90
Created by class
  Test
Calls functions
  super
  setWeeklySalary

METHOD: Test.main(String args)
  File: C:\V\ellsa_I_C_U\Test.java
Local variables
  Employee ref
  String output
  Boss b
  CommissionWorker c
  PieceWorker p
  HourlyWorker h
  DecimalFormat precision2
Calls functions
  Boss
  b.empAddress.setAddress
  CommissionWorker
  c.empAddress.setAddress
  PieceWorker
  p.empAddress.setAddress
  HourlyWorker
  h.empAddress.setAddress
  DecimalFormat
  ref.toString
  precision2.format
  ref.earnings
  b.toString
  b.earnings
  c.toString
  c.earnings
  p.toString
  p.earnings
  h.toString
  h.earnings
  JOptionPane.showMessageDialog
  System.exit
Creates Objects of Class
  CommissionWorker
  Employee
  HourlyWorker
  PieceWorker
  Boss

METHOD: CommissionWorker.CommissionWorker(String first, String last, double s, double c, int q)
  File: C:\V\ellsa_I_C_U\CommissionWorker.java
Calls functions
  super
  setSalary
  setCommission
  setQuantity

91
METHOD: CommissionWorker.setSalary(double s)
File: C:\V\ellsa_I_C_U\CommissionWorker.java
Called by Methods:
CommissionWorker

METHOD: CommissionWorker.setCommission(double c)
File: C:\V\ellsa_I_C_U\CommissionWorker.java
Called by Methods:
CommissionWorker

METHOD: CommissionWorker.setQuantity(int q)
File: C:\V\ellsa_I_C_U\CommissionWorker.java
Called by Methods:
CommissionWorker
PieceWorker

METHOD: CommissionWorker.earnings()
File: C:\V\ellsa_I_C_U\CommissionWorker.java
Called by Methods:
main

METHOD: CommissionWorker.toString()
File: C:\V\ellsa_I_C_U\CommissionWorker.java
Calls functions
super
Called by Methods:
main

METHOD: EmpInfo.EmpInfo()
File: C:\V\ellsa_I_C_U\EmpInfo.java
Called by Methods:
Employee

METHOD: EmpInfo.setAddress(String A1,String C,String S,String Z)
File: C:\V\ellsa_I_C_U\EmpInfo.java
Called by Methods:
main
main
main
main

METHOD: Employee.Employee(String first,String last)
File: C:\V\ellsa_I_C_U\Employee.java
Calls functions
EmpInfo

METHOD: Employee.getFirstName()
File: C:\V\ellsa_I_C_U\Employee.java

METHOD: Employee.getLastName()
File: C:\V\ellsa_I_C_U\Employee.java
METHOD: Employee.toString()
File: C:\V\ellsa_I_C_U\Employee.java
Called by Methods:
main

METHOD: Employee.earnings()
File: C:\V\ellsa_I_C_U\Employee.java
Called by Methods:
main

METHOD: HourlyWorker.HourlyWorker(String first,String last,double w,double h)
File: C:\V\ellsa_I_C_U\HourlyWorker.java
Calls functions
super
setWage
setHours
Called by Methods:
main

METHOD: HourlyWorker.setWage(double w)
File: C:\V\ellsa_I_C_U\HourlyWorker.java
Called by Methods:
HourlyWorker
PieceWorker

METHOD: HourlyWorker.setHours(double h)
File: C:\V\ellsa_I_C_U\HourlyWorker.java
Called by Methods:
HourlyWorker

METHOD: HourlyWorker.earnings()
File: C:\V\ellsa_I_C_U\HourlyWorker.java
Called by Methods:
main

METHOD: HourlyWorker.toString()
File: C:\V\ellsa_I_C_U\HourlyWorker.java
Calls functions
super
Called by Methods:
main

METHOD: PieceWorker.PieceWorker(String first, String last, double w, int q)
File: C:\V\ellsa_I_C_U\PieceWorker.java
Calls functions
super
setWage
setQuantity
Called by Methods:
main

METHOD: PieceWorker.setWage(double w)
File: C:\V\ellsa_I_C_U\PieceWorker.java

93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
METHOD: PieceWorker.setQuantity(int q)
File: C:\\V\ellsa_I_C_U\PieceWorker.java

METHOD: PieceWorker.earnings()
File: C:\\V\ellsa_I_C_U\PieceWorker.java
Called by Methods:
main

METHOD: PieceWorker.toString()
File: C:\\V\ellsa_I_C_U\PieceWorker.java
Calls functions
super
Called by Methods:
main

METHOD: Boss.Boss(String first,String last,double s)
File: C:\\V\ellsa_I_C_U\Boss.java
Calls functions
super
setWeeklySalary
Called by Methods:
main

METHOD: Boss.setWeeklySalary(double s)
File: C:\\V\ellsa_I_C_U\Boss.java
Called by Methods:
Boss

METHOD: Boss.earnings()
File: C:\\V\ellsa_I_C_U\Boss.java
Called by Methods:
main

METHOD: Boss.toString()
File: C:\\V\ellsa_I_C_U\Boss.java
Calls functions
super
Called by Methods:
main

OBJECT: CommissionWorker c
File: C:\\V\ellsa_I_C_U\Test.java
Actual Interface
  c.empAddress.setAddress
c.toString
c.earnings

OBJECT: Employee ref
File: C:\\V\ellsa_I_C_U\Test.java
Dynamic type(s)
  CommissionWorker
  HourlyWorker
  PieceWorker
  Boss
<table>
<thead>
<tr>
<th>Data Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>h</td>
</tr>
</tbody>
</table>

**Actual Interface**

- ref.toString
- ref.earnings
- ref.toString
- ref.earnings
- ref.toString
- ref.earnings
- ref.toString
- ref.earnings

---

**OBJECT: HourlyWorker h**

- File: C:\V\ellsa_I_C_U\Test.java

**Actual Interface**

- h.empAddress.setAddress
- h.toString
- h.earnings

---

**OBJECT: PieceWorker p**

- File: C:\V\ellsa_I_C_U\Test.java

**Actual Interface**

- p.empAddress.setAddress
- p.toString
- p.earnings

---

**OBJECT: Boss b**

- File: C:\V\ellsa_I_C_U\Test.java

**Actual Interface**

- b.empAddress.setAddress
- b.toString
- b.earnings
7.2.4 Summary of ELLSA and JFlex Validation

In addition to the validation scenarios presented in this section, JFlex was tested to verify various aspects of design, operation and output. In the course of validating JFlex, and through it the ELLSA model, several errors were discovered. During the validation process, new ideas and relationships were created as result of testing. The tracking of assignment statement data dependencies came about from the validation process as it became clear that classes were using each other, but the ELLSA had no way of presenting this fact. The Actual Interface resulted from the validation process when it became clear that maintainers are more concerned with the messages the object actually sends than the messages it could send (the Static Interface of an object).

7.3 Validation of the CIA and PIA Processes and CSM

The algorithms needed to compare software architectures were implemented in JFlex. The comparison algorithms yield a correct list of components that were modified by maintenance activities. The results produced by JFlex are identical to those obtained by manual construction. These results are readily observable from output of JFlex.

Predictive Impact Analysis (PIA) and Comparative Impact Analysis (CIA) were both implemented in JFlex. A series of tests was run on Java systems with the necessary features to require a complex impact analysis. Impact analysis performed on the modified systems yielded a list of changed components and components impacted by the changes. Verification that the changes actually occurred was done by inspection. The results listing affected or impact components are explicated on a case-by-case basis. Validation of the results was performed by inspection. Thirteen separate maintenance activities were applied to a data sample of ten Java systems. The maintenance activities were chosen on the basis of:
1. common activity
2. applicability to the system under consideration
3. modification of the object-oriented structure of the system

The third basis is required because CIA/PIA is concerned only with OO operations. CIA, for example, would not note a change in a count controlled loop condition, unless the loop referenced an object’s data field.

The maintenance activity scenarios tested are:

1. Modifying class structure by adding members and methods
2. Modifying class structure by deleting members and methods in inheritance relationship
3. Modifying class structure in contained/contains relationship
4. Modifying class structure by adding an inheritance relationship
5. Deleting a class w/ uses
6. Modifying class structure by removing an inheritance relationship
7. Modifying class structure by changing by altering a parent in an inheritance relationship
8. Modifying method structure by adding an object
9. Modifying method structure by deleting an object
10. Modifying method structure by adding a function call (tests both calls and called by)
11. Modifying method structure by deleting a function call (tests both calls and called by)
12. Changing object data dependencies
To conserve space, results are included for only three scenarios and in edited form. We have established that JFlex produces the correct ELLSA for a given set of source code. We will no longer present source code.

7.3.1 Scenario 1 – Modifying class structure

The act of adding a data member or member method is the most common maintenance activity in our experience. The generalized effects of this activity were presented in Chapter 4. In this exercise one new data member and one new method, both statically scoped, are added to class StringDemo of Scenario 1, Section 7.2.1. The ELLSAs for the original and modified systems are presented in Tables 7-6 and 7-7, respectively.

The Comparative Analysis Report (CAR) for Scenario 1 is presented in Table 7-8. By inspection the added data member and the added method are noted in the CAR. An entry is present that reports the addition of a new method component. Finally, there is an entry that describes the changes to an existing method, main. Adding a call to the newly defined method stringcopy changed the ELLSA of main.

7.3.2 Scenario 3 – Modifying Class Structure in Contained/Contains Relationship

A contained class is modified in Scenario 3. The data type of a member and a method is changed from int to double. The change is noted as a deletion of both the data member and method, and then as adding a data member and method as determined by CIA correctly. These CIA activities are described in Section 4.7. Table 7-9 shows the ELLSA of the original system in edited form. Table 7-10 shows the ELLSA of the modified system. Table 7-11 shows the CAR for Scenario 3. It is observed that the data member is reported as deleted and then adding, as for the method. There are no specific effects to the system. The contained class had one overt interaction with its
containing class, a call to the deleted method. However, the "new" method has the same name and number of parameters, and the actual parameter has not changed, so the call is still legal.

7.3.3 Scenario 9 Modifying Method Structure by Deleting an Object

In Scenario 9, the declaration an object of a used class is removed from a method. All other references to the object are left unchanged. Of course, all the references are illegal after the object is deleted. JFlex correctly locates the change, and list the impacts to the system. The classes used for this scenario are the "original" classes used in Scenario 7. The action and impacts are straightforward. Table 7 - 12 shows the TD for the modified method Test.main. The CAR is in Table 7 - 13. As listed in the CAR, all remaining references to the deleted object are now reported as illegal.

7.3.4 Scenario 12 – Changing Object Data Dependencies

In this scenario, the assignment statement data dependencies of an object are changed. Some existing dependencies are removed and some new dependencies are added. The changes are noted in the TDs for the pre- and post modified objects and methods. The dependencies were originally from a single object. In the modified system the dependencies came from two objects. Table 7 – 14 presents the CAR for Scenario 12. In this scenario, an object is added to the function "main" and is linked to the existing objects via assignment statement data dependencies.

The object's classes are not changed or impacted as a result of the changes. In the CAR for Scenario 12 JFlex notes that method Class1.main is changed. The objects are declared within Class1.main and all of the statements that contribute to the data dependencies for the object are within Class1.main. In changing the dependencies, it is necessary to change the declaring block, in this case, method Class1.main.
### Table 7-6 ELLSA of original StringDemo class

<table>
<thead>
<tr>
<th>Class: StringDemo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location: C:\V\stavich\javastn\ch02\StringDemo\StringDemo.java</td>
</tr>
</tbody>
</table>

**Data Members**

**String Demo.main(String args)**

**Static Interface**

**String Demo.main(String args)**

**Calls functions**

- sentence.indexOf
- System.out.println
- sentence.substring

---

**METHOD: StringDemo.main(String args)**

<p>| File: C:\V\stavich\javastn\ch02\StringDemo\StringDemo.java |</p>
<table>
<thead>
<tr>
<th>Local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>String sentence</td>
</tr>
<tr>
<td>int position</td>
</tr>
<tr>
<td>Calls functions</td>
</tr>
<tr>
<td>sentence.indexOf</td>
</tr>
<tr>
<td>System.out.println</td>
</tr>
<tr>
<td>sentence.substring</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
### Table 7 - 7 ELLSA for modified StringDemo class from Scenario 1

<table>
<thead>
<tr>
<th>Class: StringDemo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location: C:\V\ELLSASingleClass\new\StringDemo.java</td>
</tr>
<tr>
<td>Data Members</td>
</tr>
<tr>
<td>copy: String: Static</td>
</tr>
<tr>
<td>Member Methods</td>
</tr>
<tr>
<td>StringDemo.main(String args)</td>
</tr>
<tr>
<td>StringDemo.stringcopy(String str)</td>
</tr>
<tr>
<td>Static Interface</td>
</tr>
<tr>
<td>StringDemo.main(String args)</td>
</tr>
<tr>
<td>StringDemo.stringcopy(String str)</td>
</tr>
<tr>
<td>Calls functions</td>
</tr>
<tr>
<td>sentence.indexOf</td>
</tr>
<tr>
<td>System.out.println</td>
</tr>
<tr>
<td>sentence.substring</td>
</tr>
<tr>
<td>stringcopy</td>
</tr>
</tbody>
</table>

**METHOD:** StringDemo.main(String args)
- **File:** C:V\ELLSASingleClass\new\StringDemo.java
- **Local variables**
  - String sentence
  - int position
- **Calls functions**
  - sentence.indexOf
  - System.out.println
  - sentence.substring
  - stringcopy

**METHOD:** StringDemo.stringcopy(String str)
- **File:** C:\V\ELLSASingleClass\new\StringDemo.java
- **Called by Methods:**
  - main

---

### Table 7 - 8 Results of CIA from Scenario 1

This file contains a listing of affected components from project ellsasinglemod.jpj.
ellsasinglemod.jpj contains the following files:
C:\V\ELLSASingleClass\new\StringDemo.java
ellsasinglemod.jpj was compared to ellsasingle.jpj and the following changes and impacts were discovered:

<table>
<thead>
<tr>
<th>CLASS: StringDemo ALTERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location: C:\V\ELLSASingleClass\StringDemo.java</td>
</tr>
<tr>
<td>Added class member(s):</td>
</tr>
<tr>
<td>copy</td>
</tr>
<tr>
<td>Added class method(s):</td>
</tr>
<tr>
<td>stringcopy</td>
</tr>
</tbody>
</table>

**METHOD:** StringDemo.stringcopy ADDED in ellsasinglemod.jpj
- **Location:** C:\V\ELLSASingleClass\new\StringDemo.java

**METHOD:** StringDemo.main ALTERED
- **Location:** C:\V\ELLSASingleClass\StringDemo.java
- **Added method call(s):**
  - stringcopy

---

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
### Table 7 – 9 Partial ELLSA for original system

<table>
<thead>
<tr>
<th>Data Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>horsepower : int : Protected</td>
</tr>
<tr>
<td>manf : String : Protected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine.gethorse()</td>
</tr>
<tr>
<td>Engine.getmanf()</td>
</tr>
<tr>
<td>Engine.sethp(int hp)</td>
</tr>
<tr>
<td>Engine.setmanf(String m)</td>
</tr>
</tbody>
</table>

### Method: Engine.sethp(int hp)

- File: C:\\V\\ellsacontains\\Engine.java
- Called by Methods: Aircraft

### Object: Engine p

- File: C:\\V\\ellsacontains\\aircraft.java

### Static Interface

- gethorse: int
- getmanf: String
- sethp: void
- setmanf: void

### Actual Interface

---

### Table 7 – 10 Partial ELLSA for modified system

<table>
<thead>
<tr>
<th>Data Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>horsepower : double : Protected //changed</td>
</tr>
<tr>
<td>manf : String : Protected</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Member Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine.gethorse()</td>
</tr>
<tr>
<td>Engine.getmanf()</td>
</tr>
<tr>
<td>Engine.sethp(double hp) //changed</td>
</tr>
<tr>
<td>Engine.setmanf(String m)</td>
</tr>
</tbody>
</table>

### Method: Engine.sethp(double hp)

- File: C:\\V\\ellsacontains\\new\\Engine.java
- Called by Methods: aircraft

### Object: Engine p

- File: C:\\V\\ellsacontains\\new\\aircraft.java

### Static Interface

- gethorse: double
- getmanf: String
- sethp: void
- setmanf: void

### Data Dependencies

---

102

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 7 – 11 CAR for Scenario 3, changing a container class

This file contains a listing of affected components from project ellsacontainsmod.jpj.

ellsacontainsmod.jpj contains the following files:
- C:\V\ellsacontains\new\Engine.java
- C:\V\ellsacontains\new\aircraft.java

ellsacontainsmod.jpj was compared to ellsacontains.jpj and the following changes and impacts were discovered:

**CLASS: Engine ALTERED**
- Location: C:\V\ellsacontains\new\Engine.java
- Contained by class:
  - aircraft
- Created by class:
  - aircraft
- Object Family:
  - p declared in aircraft
- Deleted class member(s):
  - horsepower
- Deleted class method(s):
  - gethorse
- Added class member(s):
  - horsepower
- Added class method(s):
  - gethorse

**METHOD: Engine.sethp REMOVED from ellsacontains.jpj**
- Location: C:\V\ellsacontains\new\Engine.java
- Called by methods:
  - aircraft

**METHOD: Engine.sethp ADDED in ellsacontainsmod.jpj**
- Location: C:\V\ellsacontains\new\Engine.java
7.3.5 Summary of Validation of CIA, PIA and CSM

The scenarios above represent a set of common software maintenance activities that are performed on OO software. The scenarios cover all areas of concern for this research: classes, methods and object and the low-level architectural clichés that bind them together into a software system.

JFlex, and through it CSM, produced repeatedly verifiable, correct results for the systems and scenarios tested. The validation scenario systems were carefully chosen to contain the necessary interactions and relationships as to thoroughly exercise the CSM process. In the course of conducting the validation tests, several errors and omissions were discovered and corrected.

JFlex, and CIA / PIA, correctly locate and note changes and impacts to a Java software system.

7.4 Improvement of Maintenance Process

CSM automates component level impact analysis of components affected during Java program maintenance. CSM performs these useful functions:

- Generation of the ELLSA for a Java software system. The rich variety of information contained in the ELLSA tells the maintainers a great deal about the structure of the code they are maintaining. The ELLSA presents the maintainer with information on class inheritance, aggregation, association and usage.

- Predictive Impact Analysis. Maintainers can ascertain in advance what components will be affected by maintenance activities. By determining affected components before coding, resources can be allocated as needed, unforeseen side effects can be eliminated or avoided, and manual testing effort can be reduced.
Automated post maintenance impact analysis (Comparative Impact Analysis).

Impact analysis yields a list of effected components that require testing.

CSM provides the Java maintainer with an integrated methodology for evaluating and understanding OO software systems and measuring testing coverage.
Table 7 – 12 TD for method Test.main of the modified system

METHOD: Test.main(String args)
File: C:\V\CSM\ModifyParentbyadding\deleteobject\Test.java
Local variables
- DecimalFormat precision2
- String output
Calls functions
- c.getX
- c.getY
- c.getRadius
- c.getHeight
- c.setHeight
- c.setRadius
- c.setPoint
- precision2.format
- c.area
- c.volume
- JOptionPane.showMessageDialog
- System.exit

Table 7 – 13 CAR for Scenario 9

This file contains a listing of affected components from project delobjend.jpj.
delobjend.jpj contains the following files:
- C:\V\CSM\ModifyParentbyadding\deleteobject\Test.java
- C:\V\CSM\ModifyParentbyadding\deleteobject\Cylinder.java
- C:\V\CSM\ModifyParentbyadding\deleteobject\Point.java
- C:\V\CSM\ModifyParentbyadding\deleteobject\Circle.java
delobjend.jpj was compared to delobjstart.jpj and the following changes and impacts were discovered:

METHOD: Test.main ALTERED
Location: C:\V\CSM\ModifyParentbyadding\Test.java
Deleted object variable(s):
- Cylinder c

OBJECT: c REMOVED from delobjstart.jpj
Location: C:\V\CSM\ModifyParentbyadding\Test.java
- Method main still calls c.getX now ILLEGAL
- Method main still calls c.getY now ILLEGAL
- Method main still calls c.getRadius now ILLEGAL
- Method main still calls c.getHeight now ILLEGAL
- Method main still calls c.setHeight now ILLEGAL
- Method main still calls c.setRadius now ILLEGAL
- Method main still calls c.setPoint now ILLEGAL
- Method main still calls c.area now ILLEGAL
- Method main still calls c.volume now ILLEGAL

106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 7 - 14 CAR for Scenario 12

This file contains a listing of affected components from project postobjdd.jpj.
postobjdd.jpj contains the following files:
C:\V\CSM\ModifyObjDD\ObjectDDMOD\Class1.java

postobjdd.jpj was compared to preobjdd.jpj and the following changes and impacts were discovered:

METHOD: Class1.main ALTERED
Location: C:\V\CSM\ModifyObjDD\Class1.java
Deleted method call(s):
  svar.getprivar
Added method call(s):
  Math.sin
  Math.max

OBJECT: newsvar ADDED in postobjdd.jpj
Location: C:\V\CSM\ModifyObjDD\ObjectDDMOD\Class1.java
Declared in: main

OBJECT: dvar ALTERED
Location: C:\V\CSM\ModifyObjDD\Class1.java
Deleted data dependencies:
  svar.getprivar
Added data dependencies:
  newsvar.pubvar
  Math.sin

OBJECT: svar ALTERED
Location: C:\V\CSM\ModifyObjDD\Class1.java
Added data dependencies:
  Math.max

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Chapter 8

Case Studies

8.1 Introduction

We present the application of CSM to two Java software systems. The first case study shows CSM, including CIA, applied to a wide range of maintenance activities. The second case study demonstrates the ability of CSM to detect changes and determine impacts of small changes to a large system. It demonstrates the use of PIA.

8.2 Case Study 1 – Modification of a Java Linked-list Database

In this case study, we present the results of modifying a Java database system. The system, which we refer to as Student Database (SDB), records student names and test scores and calculates the student averages in some class. SDB has four classes, thirty-three methods, and five objects in four files.

SDB possesses a single pane GUI for entering, editing, deleting and displaying its data. Figure 8 – 1 shows the interface. The data are stored in a singly linked list. Each node in the list contains a field for the name, three test scores and a test score average, plus a pointer to the next node in the list. The list may be traversed only in one direction, from beginning to end.

8.2.1 Proposed Changes to the SDB GUI and Structure

SDB presents its data through the GUI in Figure 8 – 1. The GUI is to be modified in the following ways:

1. The list structure is to be converted to a doubly linked list allowing traversal in both directions.

2. A button labeled by the symbol “<” is to be added in column 2, line 7 of the GUI.

The purpose of the button is to scroll the list from end to beginning one node per
click until the start of the list is found. A click while in the start node will result in
the display of “Already at start” in a message box.

3. The name field is to be split into first and last name, with the first name field on
line one of the GUI and the last name field on line two.

![Student Scores Using One Way Lists](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Test 1</th>
<th>Test 2</th>
<th>Test 3</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 8-1 Original GUI of SDB

Making a doubly linked list out of SDB requires the addition of a “previous” pointer
to the node structure, the modification of existing method for traversal of the list
(changing pointer assignments) and the addition of new methods for activating the end to
beginning traversal direction. The new button structure requires the addition of a new
button object, creation of the button object, modification of the “action” method which
determines which button is pressed and what action to perform, and the addition of a
method to handle the action.

Converting the single name field to first name and last name requires the modification
of the class Student by deleting the single name data element and replacing it two data
elements labeled appropriately. The class is further modified by deletion of the method
that returned the single name value. Two methods replace it, one to return the first name
and one to return the last name. The method that does the assignment of data to class data
members must be modified to accept an additional string parameter and to assign that
parameter to the new data field.
Finally, data dependencies on some objects need to be modified to complete the system upgrade to a doubly linked list. To summarize, the modifications to the system require:

1. Alteration of all four classes in the system
2. Deletion of two methods
3. Addition of six methods
4. Alteration of six methods
5. Alteration of one object
6. Addition of one object

Table 8 - 1 presents the abbreviated ELLSA of the SDB in its original form. It shows only the TDs of the class components.

8.2.2 Change Impact Analysis on SDB After Modifications

The changes proposed in Section 8.2.1 were applied to a copy of the SDB system. Figure 8 - 2 shows the new GUI is shown in.

After we recompiled the system, we executed it. The output and behavior of the modified system suggested that the actual changes made to the system met the changes specified in section 8.2.1. The ELLSA for the modified system was constructed and compared to the ELLSA of the original structure. Table 8 - 2 presents the CAR for the study. It shows that all four classes were affected to level by the changes. The CAR in Table 8 - 2 also shows the modified and affected methods and objects. For most of the modified methods, changes consisted of deleting and/or adding method calls. For the objects, the data dependency of an existing object was altered.
8.2.3 Test Coverage of Case Study 1

After all modifications were completed, we recompiled and executed the system through JFlex. We added several records and deleted one. We exercised the program controls to execute as many methods as possible. As the code was executing, JFlex was creating the trace. Figure 8-3 shows the test coverage dialog for CS1. Of the twelve affected methods remaining (three were deleted), eleven executed in the sample run of the system. The one method not executed, skipped accidentally, shows that JFlex discovers affected components that are not executed. The program was subsequently executed again and all twelve affected methods were executed and found to be in the trace data.

8.2.4 Summary of Case Study 1 and Conclusions

The SDB system of CS1 contains object-oriented architectural clichés to thoroughly exercise major aspects of CIA and the ELLSA model. The changed components were noted either as directly affected or as containing or being contained in some other component that was causing either the impact or being impacted by the change.
This case study demonstrates that JFlex, the ELLSA model and CIA correctly model the architectural structure and relationships in a complex Java system, determine changed components, discover impacts throughout the system, and report both changes and impacts. With the added ability to trace dynamic execution, CSM can determine which components were re-tested in the test run. The execution trace can be viewed to help the maintainer ascertain why a method did or did not execute or execute correctly.

As the CAR indicates, all four classes, fifteen methods and two objects were altered, added or removed from the project. The removed methods were replaced methods of the same name, with new parameter structures, or by a pair of methods, as for first and last set and get methods.

Figure 8 - 3 Test coverage analysis for Case Study 1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Table 8 – 1 ELLSA for SBD classes before modification

Class: StudentTestScores
Location: C:\V\CaseStudies\Chapter15\StudentTestScores.java

Data Members
nameLabel : Label
test1Label : Label
test2Label : Label
test3Label : Label
averageLabel : Label
nameField : TextField
test1Field : IntegerField
test2Field : IntegerField
test3Field : IntegerField
averageField : IntegerField
insertButton : Button
replaceButton : Button
deleteButton : Button
blankLabel : Label
firstButton : Button
nextButton : Button
blankLabel2 : Label
countLabel : Label
countField : IntegerField
atEndLabel : Label
atEndField : TextField
students : OneWayList : Private

Member Methods
StudentTestScores.StudentTestScores()
StudentTestScores.buttonClicked(Button buttonObj)
StudentTestScores.insertStudent()
StudentTestScores.replaceStudent()
StudentTestScores.deleteStudent()
StudentTestScores.getDataOnScreen()
StudentTestScores.displayFirstStudent()
StudentTestScores.displayNextStudent()
StudentTestScores.displayCurrentStudent()
StudentTestScores.main( args)

Ancestor Classes
GBFrame

Static Interface
StudentTestScores.StudentTestScores()
StudentTestScores.buttonClicked(Button buttonObj)
StudentTestScores.main( args)

Contains classes
OneWayList

Creates objects of class
Student

Calls functions
setTitle
averageField.setEditable
countField.setEditable
atEndField.setEditable
displayCurrentStudent
insertStudent
replaceStudent
deleteStudent

113

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
table cont.

displayFirstStudent
displayNextStudent
getDataOnScreen
stud.validateData
messageBox
students.insert
students.atEnd
students.replace
students.remove
nameField.getText
test1Field.getNumber
test2Field.getNumber
test3Field.getNumber
Student
students.first
students.next
nameField.setText
test1Field.setNumber
test2Field.setNumber
test3Field.setNumber
averageField.setNumber
students.access
stud.getName
stud.getScore
stud.getAverage
countField.setNumber
students.size
atEndField.setText
StudentTestScores
frm.setSize
frm.setVisible

Class: OneWayList
Location: C:\\CaseStudies\\Chapter15\\OneWayList.java

Data Members
  first : Node : Private
  current : Node : Private
  previous : Node : Private
  size : int : Private

Member Methods
  OneWayList.OneWayList()
  OneWayList.isEmpty()
  OneWayList.atEnd()
  OneWayList.size()
  OneWayList.first()
  OneWayList.next()
  OneWayList.access()
  OneWayList.replace(Object newData)
  OneWayList.insert(Object newData)
  OneWayList.remove()

Ancestor Classes
  Object

Static Interface
  OneWayList.OneWayList()
  OneWayList.isEmpty()
OneWayList.atEnd()
OneWayList.size()
OneWayList.first()
OneWayList.next()
OneWayList.access()
OneWayList.replace(Object newData)
OneWayList.insert(Object newData)
OneWayList.remove()

Contains classes
Node
Node
Node

Container classes
StudentTestScores

Creates objects of class
Node

Calls functions
Node

---------------------------

Class: Student
Location: C:\V\CaseStudies\Chapter15\Student.java

Data Members
- NUM_TESTS : int : Final : Public : Static
- MIN_SCORE : int : Final : Private : Static
- MAX_SCORE : int : Final : Private : Static
- name : String : Private
- tests : int : Private

Member Methods
Student.Student()
Student.Student(String nm, int t)
Student.Student(Student s)
Student.setName(String nm)
Student.getName()
Student.setScore(int i, int score)
Student.getScore(int i)
Student.getAverage()
Student.getHighScore()
Student.toString()
Student.validateData()

Ancestor Classes
Object

Static Interface
Student.Student()
Student.Student(String nm, int t)
Student.Student(Student s)
Student.setName(String nm)
Student.getName()
Student.setScore(int i, int score)
Student.getScore(int i)
Student.getAverage()
Student.getHighScore()
Student.toString()
Student.validateData()

Object Family

115

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
File: C:\V\CaseStudies\Chapter15\StudentTestScores.java
stud in Method: StudentTestScores:getDataOnScreen
File: C:\V\CaseStudies\Chapter15\StudentTestScores.java
stud in Method: StudentTestScores:displayCurrentStudent
File: C:\V\CaseStudies\Chapter15\StudentTestScores.java
s in Method: Student:Student
File: C:\V\CaseStudies\Chapter15\Student.java
Creates objects of class
Student
Created by class
StudentTestScores
Student
Calls functions
Math.round
Math.max
getAverage
name.equals
-----------------------------------------------
Class: Node
Location: C:\V\CaseStudies\Chapter15\Node.java
Data Members
  data : Object : Public
  next : Node : Public
Member Methods
  Node.Node()
  Node.Node(Object theData)
Ancestor Classes
  Object
Static Interface
  Node.Node()
  Node.Node(Object theData)
Object Family
  newNode in Method: OneWayList:insert
File: C:\V\CaseStudies\Chapter15\OneWayList.java
Contains classes
  Node
Container classes
  OneWayList
  OneWayList
  OneWayList
  Node
Created by class
OneWayList
Table 8-2 CAR for SBD original compared to SDB modified

This file contains a listing of affected components from project CSlnew.jpj.

CSlnew.jpj contains the following files:
C:\\V\CaseStudies\Chapter15\mod\StudentTestScores.java
C:\\V\CaseStudies\Chapter15\mod\OneWayList.java
C:\\V\CaseStudies\Chapter15\mod\Student.java
C:\\V\CaseStudies\Chapter15\mod\Node.java

CSlnew.jpj was compared to CSlold.jpj and the following changes and impacts were discovered:

CLASS: StudentTestScores ALTERED
Location: C:\\V\CaseStudies\Chapter15\StudentTestScores.java
Contains class:
OneWayList
Added class member(s):
  LnameLabel
  FNameField
  prevButton
Added class method(s):
  displayPrevStudent

CLASS: OneWayList ALTERED
Location: C:\\V\CaseStudies\Chapter15\OneWayList.java
Contains class:
  Node
  Node
  Node
Contained by class:
  StudentTestScores
Added class method(s):
  atStart
  prev

CLASS: Student ALTERED
Location: C:\\V\CaseStudies\Chapter15\Student.java
Created by class:
  StudentTestScores
  Student
Object Family:
  stud declared in insertStudent
  stud declared in getDataOnScreen
  stud declared in displayCurrentStudent
  s declared in Student
Deleted class member(s):
  name
Deleted class method(s):
  getName
Added class member(s):
  fName
  lname
Added class method(s):
  getName
  getiName

117

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CLASS: Node ALTERED
Location: C:\V\CaseStudies\Chapter15\Node.java
Contains class:
  Node
Contained by class:
  OneWayList
  OneWayList
  OneWayList
  Node
Created by class:
  OneWayList
Object Family:
  newNode declared in insert
Added class member(s):
  prev

METHOD: Student.Student REMOVED from CS1old.jsp
Location: C:\V\CaseStudies\Chapter15\Student.java
Creates classes:
  Student

METHOD: Student.setName REMOVED from CS1old.jsp
Location: C:\V\CaseStudies\Chapter15\Student.java

METHOD: Student.getName REMOVED from CS1old.jsp
Location: C:\V\CaseStudies\Chapter15\Student.java
Called by methods:
  displayCurrentStudent

METHOD: StudentTestScores.displayPrevStudent ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\StudentTestScores.java

METHOD: OneWayList.atStart ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\OneWayList.java

METHOD: OneWayList.prev ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\OneWayList.java

METHOD: Student.Student ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\Student.java

METHOD: Student.setName ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\Student.java

METHOD: Student.getfName ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\Student.java

METHOD: Student.getlName ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\Student.java

METHOD: StudentTestScores.buttonClicked ALTERED
Location: C:\V\CaseStudies\Chapter15\StudentTestScores.java
Added method call(s):
  displayPrevStudent

118

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
METHOD: StudentTestScores.getDataOnScreen ALTERED
Location: C:\V\CaseStudies\Chapter15\StudentTestScores.java
Added method call(s):
   lnameField.getText

METHOD: StudentTestScores.displayCurrentStudent ALTERED
Location: C:\V\CaseStudies\Chapter15\StudentTestScores.java
Deleted method call(s):
   stud.getName
Added class variable(s):
   Node
Added method call(s):
   lnameField.setText
   stud.getfName
   stud.getlName

METHOD: OneWayList.remove ALTERED
Location: C:\V\CaseStudies\Chapter15\OneWayList.java
Deleted method call(s):
   name.equals
Added class variable(s):
   Node
Added method call(s):
   lname.equals
   fname.equals

METHOD: Student.validateData ALTERED
Location: C:\V\CaseStudies\Chapter15\Student.java
Deleted method call(s):
   name.equals
Added method call(s):
   lname.equals
   fname.equals

OBJECT: temp ADDED in CS1new.jsp
Location: C:\V\CaseStudies\Chapter15\mod\OneWayList.java
Declared in: remove

OBJECT: newNode ALTERED
Location: C:\V\CaseStudies\Chapter15\OneWayList.java
Declared in: insert
Added data dependencies:
   previous
8.3 Case Study 2 – Modifications to a Large Java System

This case study presents a series of modifications to a large Java software system. The source is the “zip” packages of Sun's Java distribution. Typically, it is found as a subdirectory to the “util” package. The package consists of 19 classes, 170 methods and 16 internal objects in 18 files.

The ELLSA for this case study is lengthy; therefore, we present only pertinent sections when needed.

In CS2, we perform PIA on the “zip” system. We introduce a series of changes in the system. One change is performed for each type of change possible for PIA. They are:

1. Delete a data member
2. Add a data member
3. Delete a method
4. Add a method
5. Add a class
6. Modify a class (change extends and implements)
7. Modify a method (change calls to methods)
8. Delete an object

The changes are performed as a group with a single overall result displayed.

8.3.1 Modifying “zip” Source Code

The changes applied to “zip” are not intended to represent connected maintenance activities. They are simply a series of unrelated, individual activities. CS2 is designed to simulate conditions under which the code might be modified by a group of programmers working on different aspects of the system. We applied the following changes:

1. Protected data member “inf” of type Inflater was deleted from class
InflaterInputStream; deleted protected member def of type Deflator in class DeflaterOutputStream.

2. Private data member “size” of type long was added to class ZipFileInputStream; added new member “newdata” of type String to class Adler32.

3. Deleted method getValue() of class Checksum; deleted method InflaterInputStream.fill().

4. Added method readHeader(long pos) to GZIPInputStream, with local object CheckedInputStream in and calls to crc.reset and readUShort; added method getSize() to ZipInputStream.

5. Added class “Newclass” extending ZipFile and implementing ZipConstants; added class “ZIPClass” extending nothing and implementing ZipConstants.

6. Modified class ZipOutputStream to extend nothing and implement nothing; modified class CheckedInputStream by changing extends from FilterInputStream to Adler32.

7. Modified method ZipOutputStream.closeEntry() by removing a method call (def.finish); removed method call update1 from CRC32.update().

8. Deleted object ZipEntry e from method ZipOutputStream.WriteCEN(ZipEntry e); deleted object cksum from method CheckedInputStream.CheckedInputStream().

8.3.2 Results and Conclusions of Case Study 2

After completion of the changes, the ELLSA for the virtual system was created. It was compared to the ELLSA of the original system. Table 8 – 3 presents the CAR resulting from that comparison. The CAR notes all of the changes listed above along with components affected by the changes.
The two new classes are noted first. There are no impacts from these additions. The next eight entries describe the changes and impacts from modifying classes. The two deleted class data members are noted as are the two deleted methods. Two changes in inheritance structure are noted as well. Two added class members are noted, one each in ZipFileInputStream and Adler32. The two added methods are described as additions to their respective class and as component level addition as methods. The CAR also notes the two deleted methods as a component. Four methods were altered in the course of this exercise. Two lost object declarations and two lost method calls. All are correctly noted. Finally, two objects were removed from the system. Their absence is noted, along with the fact that one of the containing methods is still trying to pass messages to its deleted object.

For this large scale Java program, JFlex, PIA and CSM correctly parsed the code, created the ELLSA, accepted the "virtual" changes to the system and created a listing of changed, impacted and affected components. This case study demonstrates the ability of JFlex and CSM to handle correctly its intended functions.

8.4 Summary

We have shown two case studies to demonstrate the feasibility and reliability of CSM and JFlex. JFlex successfully created ELLSA structures for a large Java system of 170 methods and 19 classes. CSM recognized changes in a system and generated correct impact analysis on the changes. Test coverage analysis is not possible using PIA. A "virtual" system has only an interface. It has no code.
cs2new2.jpj contains the following files:

- C:\\CaseStudies\\zip\\Adler32.java
- C:\\CaseStudies\\zip\\CheckedInputStream.java
- C:\\CaseStudies\\zip\\Checksum.java
- C:\\CaseStudies\\zip\\CRC32.java
- C:\\CaseStudies\\zip\\DataFormatException.java
- C:\\CaseStudies\\zip\\InflaterInputStream.java
- C:\\CaseStudies\\zip\\DeflaterOutputStream.java
- C:\\CaseStudies\\zip\\GZIPInputStream.java
- C:\\CaseStudies\\zip\\GZIPOutputStream.java
- C:\\CaseStudies\\zip\\Inflater.java
- C:\\CaseStudies\\zip\\Deflater.java
- C:\\CaseStudies\\zip\\ZipOutputStream.java
- C:\\CaseStudies\\zip\\ZipEntry.java
- C:\\CaseStudies\\zip\\ZipException.java
- C:\\CaseStudies\\zip\\ZipFile.java
- C:\\CaseStudies\\zip\\ZipInputStream.java
- C:\\CaseStudies\\zip\\ZipConstants.java

cs2new2.jpj was compared to cs2old.jpj and the following changes and impacts were discovered:

CLASS: NewClass ADDED in cs2new2.jpj
Location:

CLASS: ZIPClass ADDED in cs2new2.jpj
Location:

CLASS: Adler32 ALTERED
Location: C:\\CaseStudies\\zip\\Adler32.java
Added class member(s):
newdata

CLASS: CheckedInputStream ALTERED
Location: C:\\CaseStudies\\zip\\CheckedInputStream.java
Class parent was FilterInputStream now has parent Adler32
Contains class:
Checksum
Created by class:
GZIPInputStream
Object Family:
in declared in readHeader

CLASS: Checksum ALTERED
Location: C:\\CaseStudies\\zip\\Checksum.java
Contained by class:
CheckedInputStream
CheckedOutputStream
Created by class:
CheckedInputStream
CheckedOutputStream

123
Object Family:
  checksum declared in CheckedInputStream
  checksum declared in CheckedOutputStream
Deleted class method(s):
  getValue

CLASS: InflaterInputStream ALTERED
Location: C:\WNCaseStudies\zip\InflaterInputStream.java
Descendant classes:
  GZIPInputStream
  ZipInputStream
Contains class:
  Inflater
Deleted class member(s):
  inf
Deleted class method(s):
  fill

CLASS: DeflaterOutputStream ALTERED
Location: C:\WNCaseStudies\zip\DeflaterOutputStream.java
Descendant classes:
  GZIPOutputStream
  ZipOutputStream
Contains class:
  Deflater
Deleted class member(s):
  Def

CLASS: GZIPInputStream
Location: C:\WNCaseStudies\zip\GZIPInputStream.java
Contains class:
  CRC32
Added class method(s):
  readHeader

CLASS: ZipOutputStream ALTERED
Location: C:\WNCaseStudies\zip\ZipOutputStream.java
Class is now Orphan – parent DeflaterOutputStream deleted
Contains class:
  ZipEntry
  CRC32

CLASS: ZipFileInputStream ALTERED
Location: C:\WNCaseStudies\zip\ZipFile.java
Contains class:
  ZipFile
  ZipEntry
Added class member(s):
  size

CLASS: ZipInputStream ALTERED
Location: C:\WNCaseStudies\zip\ZipInputStream.java
Contains class:
  ZipEntry
  CRC32
table cont.

Added class method(s):
   getSize

METHOD: Checksum.getValue REMOVED from cs2old.jpj
   Location: C:\V\CaseStudies\zip\Checksum.java

METHOD: InflaterInputStream.fill REMOVED from cs2old.jpj
   Calls methods:
      in.read
      EOFException
      setInput
   Called by methods:
      read

METHOD: GZIPInputStream.readHeader ADDED in cs2new2.jpj
   Location: MEMORY

METHOD: ZipInputStream.getSize ADDED in cs2new2.jpj
   Location: MEMORY

METHOD: CheckedInputStream.CheckedInputStream ALTERED
   Deleted object variable(s):
      Checksum

METHOD: CRC32.update ALTERED
   Deleted object variable(s):
      ZipEntry
   Deleted method call(s):
      update1

METHOD: ZipOutputStream.closeEntry ALTERED
   Deleted object variable(s):
      ZipEntry
   Deleted method call(s):
      def.finish

METHOD: ZipOutputStream.writeCEN ALTERED
   Deleted object variable(s):
      ZipEntry

OBJECT: checksum REMOVED from cs2old.jpj
   Location: C:\V\CaseStudies\zip\CheckedInputStream.java

OBJECT: e REMOVED from cs2old.jpj
   Location: C:\V\CaseStudies\zip\ZipOutputStream.java

   Method writeCEN still calls e.name.length now ILLEGAL
   Method writeCEN still calls e.comment.length now ILLEGAL
Chapter 9

Summary, Contributions, Extensions and Conclusion

9.1 Summary

In Chapter 1, we described the problems inherent to OO software maintenance. The reader will recall them as the understanding, complex dependency, and tool support problems. Through this research, we have addressed these problems in the following ways.

- This research helps alleviate the understanding problem through the creation of a methodology capable of detecting and recording relationships and interactions of OO systems. The Extended Low Level Software Architecture model provides detailed information on classes, methods and objects. The ELLSA gives the maintainer an understanding of an OO system’s component.
- This research helps address the complex dependency problem by providing a data model designed specifically to capture OO interactions such as inheritance, aggregation, association, uses/used relationships, and complex data dependencies. It improves impact analysis by refining the granularity of class, method and object analysis. It provides a detailed process for understanding how the effects of change ripple through OO systems.
- This research helps address the tool support problem by automating the CSM process in JFlex. Knowledge support for Java software maintenance is readily obtainable from JFlex.

We have presented the Comparative Software Maintenance methodology. Designed specifically for OO software, CSM presents the maintainer with knowledge of the types...
of relationships and interactions that make understanding OO systems difficult. CSM allows the maintainer to predict how change will affect the system, and to review how the system has changed over time.

9.2 Conclusion

We have developed an OO maintenance methodology based upon the architectural structures found in OO systems. The original hypothesis of this work was that the architectural clichés found in OO systems could be used to predict (pre-modification) or to determine (post-modification) the impact of proposed or actual system changes. We developed PIA for pre-modification impact analysis and CIA for post-modification impact analysis. Both forms of impact analysis are based upon the architecture of the systems they are analyzing. CSM carries the results of impact analysis and ripple effect analysis through to the retesting of affected and changed components.

A goal of this research was to determine ways in which the architectural structure of an object-oriented software system affect the maintenance of that system and to investigate if that structure could be used as an aid to program comprehension and maintenance rather than a hindrance to it. The contributions of this research are summarized in section 9.2.1. Extensions of and future work in this research are presented in section 9.2.2.

9.2.1 Contributions

The main contributions of this research fall into two categories: program comprehension and impact analysis. For program comprehension, we developed the ELLSA and its implementation in JFlex. For impact analysis, we improved existing processes and implemented the processes to perform automated impact analysis of Java software systems. This approach identified new relationships and interactions that play a role in program understanding and maintenance. New relationships and interactions
include the static data interface of a class, the assigned from data dependencies for object and classes, and the actual interface of an object.

The ELLSA model provides detailed knowledge of an OO software system. As a model, ELLSA discovers and records relationships and interactions such as inheritance, containment, association and messaging in an OO system that contribute to the architectural structure of the system. Understanding these relationships and interactions is necessary for program understanding. These relationships provide a means for evaluation of change in an OO system. The ELLSA model allows the maintainer to understanding the nature of changes to the system and their impacts. Automation of the model and graphical representation of the results gives the maintainer insights and views of the system that allow he or she to quickly determine what components need to be changed and to assess the impact of those changes.

This research makes the following contributions to the field of object oriented software maintenance and testing:

1. Extends the LLSA data model data – New analysis is possible using the basic model plus the new data extensions. The new data include the number, order and type of the class members, local variables, and function parameters. The data were, in general, not necessary for creating the LLSA as a model for understanding the system's structure; but, it is required in order to perform comparative maintenance, impact analysis and testing of modified software.

2. Defines new algorithms for generating architectural information. These new algorithms help to perform comparative maintenance, impact analysis and testing of modified software.
3. Defines new algorithms for comparing architectural information. These algorithms give the ability to perform the set operations \textit{difference} and \textit{intersection} on two separate software components. The modified components can than be automatically detected.

4. Improves software change impact analysis – The impact analysis performed here contains more information about components than previous work. Instead of providing a simple description such as “X inherits from Y”, CSM supplies details of interactions. It provides such data as which data members and methods are inherited, which methods and objects link one class to another, and what data dependencies exist between classes. CSM provides more relationships than previous data models used for impact analysis. The ELLSA records data dependencies for objects, details on data members and their relationships with subclasses and data on how relationships between classes and methods are formed by either parameter declaration or local variable declaration. CSM fully automates CIA.

5. Provides predictive information – predictive impact analysis gives the software maintainer the ability to measure the effects of maintenance activities before they are actually committed.

6. Provides automated tool support – JFlex embodies the comparative features that are useful for analyzing OO software both, statically and dynamically. It provides the automated tool support necessary to perform CSM.

Table 9 - 1 represents the summary of IA described in Chapter 2 with the addition of PIA and CIA. It highlights how PIA and CIA improves upon previous approaches.
Table 9 – 1 Comparison of IA approaches including CIA and PIA

<table>
<thead>
<tr>
<th>Approach</th>
<th>Model</th>
<th>Automated</th>
<th>Granularity</th>
<th>Predictive</th>
<th>COMparative</th>
<th>Display</th>
<th>Source</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOTME</td>
<td>ORD</td>
<td>Semi</td>
<td>Medium</td>
<td>Yes</td>
<td>Low</td>
<td>Low</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Omega</td>
<td>C++D</td>
<td>No</td>
<td>Multiple</td>
<td>No</td>
<td>NA</td>
<td>No</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>Algorithmic</td>
<td>NA</td>
<td>No</td>
<td>Fine</td>
<td>No</td>
<td>NA</td>
<td>Low</td>
<td>None</td>
<td>Code</td>
</tr>
<tr>
<td>IAS</td>
<td>Ukwn</td>
<td>Semi</td>
<td>Coarse</td>
<td>Yes</td>
<td>Low</td>
<td>No</td>
<td>Low</td>
<td>Documentation,</td>
</tr>
<tr>
<td>Visual IA</td>
<td>Ukwn</td>
<td>No</td>
<td>Coarse</td>
<td>Yes</td>
<td>Low</td>
<td>No</td>
<td>Ukwn</td>
<td>Code</td>
</tr>
<tr>
<td>Coupling</td>
<td>NA</td>
<td>No</td>
<td>Coarse</td>
<td>Yes</td>
<td>Low</td>
<td>No</td>
<td>NA</td>
<td>Metrics</td>
</tr>
<tr>
<td>CIA</td>
<td>LLSA</td>
<td>Full</td>
<td>Fine</td>
<td>No</td>
<td>NA</td>
<td>Yes</td>
<td>High</td>
<td>Architecture</td>
</tr>
<tr>
<td>PIA</td>
<td>LLSA</td>
<td>Semi</td>
<td>Fine</td>
<td>Yes</td>
<td>High</td>
<td>No</td>
<td>High</td>
<td>Architecture</td>
</tr>
</tbody>
</table>

9.2.2 Possible Future Work and Extensions to the Research

There are open problems related to numerous areas of this research. Based upon the nature of OO systems, we could investigate whether a series of maintenance activities applied to a set of systems with similar architectural clichés produces the same impact in every instance. In other words, do standard maintenance activities cause “patterns” of impacts? In order to do so we must compare two systems to determine if they are similar enough to be considered as having the “same” architecture. CSM already has the capability to compare two systems. Extending it to compare multiple systems is possible. This research would require a measurement of “sameness”.

From a different perspective, we could use CSM as a tool for tracking system evolution. This tracking could be done retrospectively or with current system upkeep. JFlex produces a list of changed components. This process could be modified to allow amendments to the list.
A modified CSM could generate version change documentation. JFlex already generates data about changes and impacts. The output of JFlex could provide "prose" descriptions of the changes and impacts. The ELLSA contains a rich body of knowledge about an OO system. This knowledge could generate documentation about changes to the system.
Bibliography


136

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Vita

Michael Alan Hoffman was born in New Orleans, Louisiana. He has earned a bachelor of arts in philosophy from the University of New Orleans and an associate of science in computer science from Southeastern Louisiana University. After teaching high school computer science courses for several years, he entered graduate school at the University of Southern Mississippi, earning a master of science degree in computer science in 1992. He was an instructor of computer science at Louisiana State University in Baton Rouge for three years. At the Fall, 2000, commencement, he will be granted the degree of Doctor of Philosophy from Louisiana State University in Baton Rouge. He is currently employed as a software engineer in Seattle, Washington. He has a beautiful wife, Fei, and an adorable son, Sean, and another little one on the way.
Candidate: Michael Alan Hoffman

Major Field: Computer Science

Title of Dissertation: A Methodology to Support the Maintenance of Object-Oriented Systems Using Impact Analysis

Approved:

[Signatures]

Dean of the Graduate School

EXAMINING COMMITTEE:

[Signatures]

Date of Examination:

October 31, 2000