RE
513
c.1

Measuring Ripple Effect for Object-Oriented Programs

by

Hani Hassan Salem

89155

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

Thesis Advisor: Dr. Nashat Mansour

Division of Computer Science & Mathematics
LEBANESE AMERICAN UNIVERSITY
July 2004

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

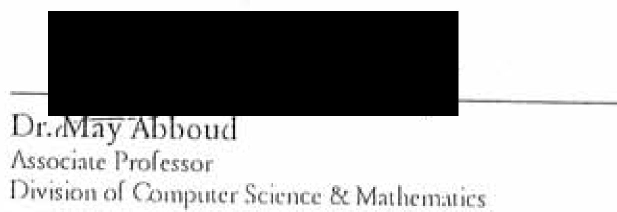
We hereby approve the thesis of

Hani Hassan Salem

Candidate for the *Master of Science* degree*.

Dr. Nashat Mansour
Associate Professor
Division of Computer Science & Mathematics

Dr. Ramzi A. Haraty
Associate Professor
Division of Computer Science & Mathematics



Dr. May Abboud
Associate Professor
Division of Computer Science & Mathematics

*We also certify that written approval has been obtained for any proprietary material contained therein.

I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.

Measuring Ripple Effect for Object-Oriented Programs

Abstract

by

Hani Hassan Salem

Ripple effect is a measure of structural complexity of a source code upon changing a method or a class. Ripple effect measures the amount by which this method / class may affect other methods or classes within a program, or programs within a system, if changes are made. Measurement of ripple effect has been incorporated into several software maintenance models because it shows maintainers the ramifications of any change that they may make before that change is actually implemented. Thus, computation of ripple effect provides a potentially valuable source of information. In this thesis, we propose a ripple effect measure for object-oriented programs and use it to compute an index for logical stability. Our approach is based on a new algorithm that will calculate the ripple effect for object-oriented programs at the code level by calculating both intra-class propagation and inter-class propagation for each class. It also determines the architecture ripple effect at the system level. Our method is based on matrix arithmetic for producing a ripple effect and logical stability measures and is illustrated by applying it to two examples.

To Mom & Dad

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Nashat Mansour for his guidance and helpful hints throughout my M.S studies. Thanks also to Dr. Ramzi Haraty and Dr. May Abboud for being on my Thesis committee.

I would like also to thank Dr. Sue Black from South Bank University for her encouragement and trust.

Finally, I would like to thank my family and best friend friends for their long support.

Table of Contents

Page

1.Introduction.. 8

2. Background on Ripple effect and logical stability 15

2.1 Ripple Effect as a complexity measure 15

2.1.1 Code Level Logical Stability 16

2.1.2 Design Level Logical Stability.............................. 18

2.2 Ripple Effect and Software Maintenance 18

2.3 Automating Ripple Effect 20

2.3.1 Tools Which Produce a Ripple Effect Measure.......... 21

2.3.2 Tools Which Trace Ripple Effect through a System...... 22

2.3.3 Tools which Trace Impact Analysis and Ripple Effect through Object Oriented Paradigm 24

2.3.4 Other 26

3. Object Oriented Dependences and Relations 28

3.1 Change Impact Definitions 31

3.2 Object-oriented System Dependency 32

3.3 Types of Changes and Their Relationship 34
3.4 Object Oriented Features and Relationships 36

3.4.1 Encapsulation.. 36
3.4.2 Inheritance.. 37
3.4.3 Polymorphism 37
3.4.4 Dynamic Binding.. 37

3.5 Change Propagation.. 37
3.5.1 Intra Class change propagation............................ 38
3.5.2 Inter-class change propagation 38
3.5.3 System dependency propagation........................... 39

3.6 Metrics for Object Oriented Ripple Effect......................... 39
3.6.1 Inter-Class Metrics 39
3.6.2 Intra-Class Metrics.. 41

4. Counting Ripple Effect as a Matrix Product..... 42

4.1 Intraclass Change Propagation 42
4.1.1Decomposition of Matrix *Zm*.................................. 46
4.1.2 Problems in calculating inter-class propagation 50

4.2 Interclass Change Propagation 51

4.3 Complexity and Logical Stability 53

4.4 Architecture Level Ripple Effect 55

5. Examples 58

5.1 Computing Ripple Effect for an Example Class.. 58

5.2 Counting Ripple effect for Calculations.java class 67

6. Future Work 72

6.1 Ripple Effect in Distributed Systems....................................... 73

6.2 Ripple Effect and Regression Testing....................................... 74

6.3 Ripple Effect Using UML.. 74

6.4 Possible Class-Level Ripple Effect Analysis estimation with syntactic impact 75

6.5 Ripple Effect in Web Applications 78

7. Conclusion 79

7.1 Concluding Remarks..... 80

Reference 81

List of Figures

Figure	Title	Page
Figure 1.1	A methodology for software maintenance	11
Figure 4.1	Example code of Class C	43
Figure 4.2	Assignment and Definition/ use	47
Figure 4.3	Assignment and Definition/ use information held in Matrix B	48
Figure 5.1	Example.java	59
Figure 5.2	Calculation.java	71

List of Tables

Figure	Title	Page
Table 2.1	Table comparing speed and accuracy of logical stability computation	22
Table 5.1	REA results	65
Table 6.6	Attribute (A) of Class (C)	76
Table 6.7	Method (M) of Class (C)	76

Chapter 1

Introduction

Ripple effect has an intuitive appeal. Imagine a stone being thrown into a pond: it makes a sound as it enters the water and causes ripples to move outwards to the edge of the pond. It is reasonably easy to transfer this image to source code. The stone entering the water is now a hypothetical change to the source code of a program, the effect of the change ripples across the source code via data flow.
As part of software development or maintenance we may want to ask questions about the program such as: How much ripple is there? Which parts of the program affect other parts the most?

The Ripple-effect measure has been identified as valid and necessary within several software maintenance models, particularly the SADT model [PB90] and the Methodology for Software Maintenance [YC80][Ben90]. Typically, 70% of software development budgets are spent on maintenance, thus its importance in the field of software engineering cannot be denied. Any measures or tools which can assist maintainers in their role by speeding up the rate at which changes can be made, or enabling maintainers to make better informed decisions on code changes can thus make an important contribution. Software maintenance was originally classified by Swanson in 1976 into three types [SW76]:

- Corrective maintenance: to address processing, performance or implementation failure.
- Adaptive maintenance: to address change in the data or processing environments.
- Perfective maintenance: to address processing efficiency performance enhancement and maintainability.

The classification was redefined by the IEEE glossary [IE90] in 1990 to include:

- Prevention maintenance: to address activities aimed at increasing the system's maintainability.
- Maintenance types are divided into corrections that correct a defect, and enhancements that implement a change to the system which changes the behavior or implementation of the system.

As all types of maintenance involve making changes to source code, ripple effect can be used to help maintainers by highlighting modules which may cause problems during the maintenance process. Ripple effect can show the maintainer what the effect of that change will be on the rest of the program or system. It can highlight classes with high ripple effect as possible problem classes, show the impact in terms of increased ripple effect or look at the ripple effect of a program and its classes before and after a change to ascertain whether the change has increased, or perhaps decreased, the stability of the program. Maintenance is difficult [vZE93] because it is not clear where modifications have to be made or what the impact will be on the rest of the source code once those changes are made; the ripple effect can certainly be used to help maintainers with the latter. Ripple effect, is not the answer to all maintainers' problems, but can be used as part of a suite of metrics it can give maintainers useful information to make their task easier.

There is a strong link between software maintenance and ripple effect. Computation of ripple effect and logical stability of a class are based on a subset of maintenance activity: a change to a single variable definition within a class. Regardless of the complexity of the maintenance activity being performed, maintenance fundamentally consists of modifications to variables within classes of code. Logical stability is computed based on the impact of these modifications. It can be used to predict the impact of primitive modifications on a program, and so be used to compute the logical stability of classes with respect to those primitive modifications. The effect of modification may not be local to the class but may affect other parts of the program. Therefore, there is a ripple effect from the location of the modification to other parts of the program that are affected by the modification. If the logical stability of a

program is poor, then the impact of any modification is large and hence the maintenance cost will be high and reliability may also suffer.

Several software maintenance models have been proposed in the past. Boehm's model [Boe87] consists of three major phases:

- Understanding the software;
- Modifying the software;
- Revalidating the software.

There are fundamentals activities of the software maintenance process. With Yau's model, A Methodology for Software [YC80], impact analysis is introduced into the lifecycle. The model consists of four phases, and includes analysis and monitoring of the impact of change at phase three accounting for ripple effect (see Figure 1.1). The aims of the model are to assist in achieving cost effective software maintenance and the development of easily maintainable software. Phase one is the analyzing the program in order to understand it. The complexity of the program, the documentation and the self-descriptiveness of the program contribute to the ease of understanding of the program. Phase two is generating a particular maintenance proposal to accomplish the implementation of the maintenance objective. The third phase is accounting for all of the ripple effect as a consequence of program modifications. The effect may not be local to the modification but may also affect other portions of the program. The main attribute affecting the ripple effect a as a consequence of a program modification is the stability of a program, that is the resistance to the amplification of changes in a program, the fourth phase the modification program is tested to ensure that it has at least the same reality as before.

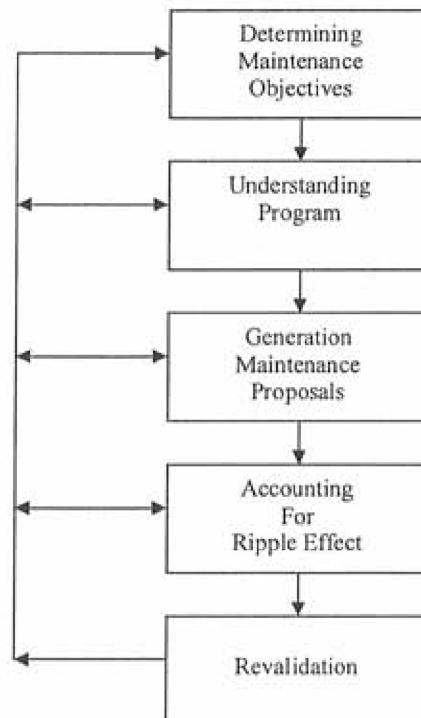


Figure 1.1 A methodology for software maintenance

The Peeger and Bohner model[PB90], SADT Diagram of Software Maintenance Activities has six phases, the main difference from Yau's model being that it includes analysis software change impact at phase two, i.e. much earlier in the life cycle. The feedback paths in the SADT model indicate attributes that must be measured; the results are ten assessed by management before the next activity is undertaken. The metrics act as a controlling mechanism in the progression from existing system and change requests to new system. Manage software maintenance controls the sequence of activities by receiving feedback and determining the next appropriate action. *Analyze software change impact* evaluates the effects of a proposed change; it determines if change can be made without disturbing the rest of the software. *Understand software under change* involves source code and related analysis, i.e. of documentation to understand the system and the proposed change. *Implement maintenance change* generates the proposed change. *Account for ripple effect* analyses the propagation of changes to other modules as a result of the change just implemented. The modifications are tested to meet new requirements and the overall system in subject to regression testing in the last phase: *Test affected software*.

A method for computing ripple effect was developed early by Yau and Collofello [YCM78] and developed over several years. It was proved difficult to write simple software using this algorithm; ripple effect tools have either taken an excessive amount of time to produce ripple effect measures or needed some user intervention to make critical decisions about the source code. Previous tools developed to produce ripple effect measures for procedural software have used Yau and Collofello's algorithm which is based on set theory. Several attempts have been made at using the algorithm to construct a tool to produce fast and accurate ripple effect measurements, none of which have completely succeeded. Black [Bla01] had reformulated the ripple effect algorithm using matrix arithmetic and used this reformulated algorithm to produce a software tool: REST [Bla01]. However, Black's algorithm computes ripple effect only for procedural programs.

Several works have been done concerning the object-oriented programs. However, these studies discussed only the change impact analysis for object-oriented paradigm and its relation with regression testing but none mentioned or suggested a tool or measure for the ripple effect in object-oriented programs. Kung et al. [KGH94] were interested in the system-wide impact of changes for regression-testing purposes. Li and Offutt [LO96] presented algorithms to analyze the potential impacts of changes to object-oriented software, taking into account encapsulation, inheritance and polymorphism. Kabaili et al [KKLO2] used a model to calculate the impacted classes due to an atomic change. They presented an extension of the changed impact model; the first extension took into account the classes that are impacted by ripple effect, whereas, in the second extension, they counted classes that need to be re-tested even if they are not directly affected. Barbara Ryder and Frank Tip [RT01] used a call graph for change impact analysis. Further on change impact analysis for object oriented programs can be found in: [CKL99], [RAJ01], [RT01], [WK00] [BLS02], [KGH94], [LOA00], [LI98], [LO96].

In this thesis we propose ripple effect and logical stability measures for object-oriented program using matrix arithmetic. We provide an analysis of object-oriented dependencies, relations, and propagations inside and outside classes. Also, we study object-oriented complexity metrics and their relation to ripple effect and classify them

as inter-class metrics and intra-class after introducing new object-oriented metrics. We calculate the ripple effect for object-oriented programs at the code level. Both intra-class propagation and inter-class propagation for each class are considered. We also calculate the architectural ripple effect at the system level. Our algorithm also clarifies the process of computing ripple effect. Each matrix used within the algorithm holds a particular type of information about the software under scrutiny. This makes it easier to understand what each part of the algorithm means and how the ripple effect is being computed.

This thesis is organized as follows:

Chapter 2 Review the literature about ripple effect and logical stability measures. Ripple effect is included in several software maintenance models as part of the maintenance lifecycle. We take a look at these models and show where ripple effect measurement fits in.

Chapter 3 gives a detailed description of all the object-oriented dependencies and relations. A list is made describing Object-oriented System dependencies which are categorized as: Class-to-Class dependencies, Class to Method, Class to Variable, Method to Variable and Method to Method. Also, all the possible types of changes made at the system level and class level are listed and their Relationship. After that, we describe object-oriented features such as inheritance, polymorphism, aggregation and data binding and their relation with ripple effect analysis and list inter-class and intra-class metrics. Finally, we study object-oriented complexity metrics and classify them into intra-class and inter-class metrics and study each one relation with ripple effect.

Chapter 4 gives a precise definition of the computation of ripple effect in Object Oriented software. Two fundamental ideas in its computation are code level which includes intra-class and inter-class change propagation and architecture level. This

chapter gives a detailed description of what they are and how they may be computed using two matrices

Chapter 5 describes in detail how to apply our proposed approach on two example classes showing step by step computation of ripple effect and logical stability using matrix arithmetic.

Chapter 6 suggests a list of future work such as fully automating our approach, measuring ripple effect for distributed systems and web applications, measuring ripple effect using Unified Model Language (UML), the use of ripple effect in regression testing and class ripple effect analysis using syntactic impact.

Chapter 7 is the concluding chapter in which we summarize and conclude our thesis.

Chapter 2

Background on Ripple effect and logical stability

The ripple effect measures the effect that a change to a single variable or a statement will have on the rest of a program or how likely it is that a change to a particular method or class is going to cause problems in the rest of a program. It can determine the scope of the change and provide a measure of the program complexity. It can also be used as an indicator of the complexity of a particular method, class or program. Ripple effect was one of the earliest metrics concerned with the structure of a system and how its modules interact [She93].

The first mention of the term ripple effect in software engineering is by Haney in 1972 [Han72]. He used a technique called "module connection analysis" to estimate the total number of changes needed to stabilize a system. Myers [Mye80] used the joint probability of connection between all elements within a system to produce a program stability measure. A matrix is set up to store the weighting of each possible connection within a system, then another matrix is derived estimating the joint probability density for any two states in the first matrix. The limit probability vector is found using these matrices and used to calculate the stability of the system. Soong [Soo77] used the joint probability of connection of all elements within a system to produce a program stability measure. Haney, Myers and Soong's methods are all measures of probability, the probability of a change to a variable or module affecting another variable or module. Yau and Collofello's ripple effect uses similar ideas of my research however; their ripple effect is not a measure of probability.

2.1 Ripple Effect as a complexity measure

When Yau and Collofello first proposed a ripple effect analysis technique in 1978 [YCM78] they saw it as a complexity measure, which could be used during software

maintenance to evaluate and compare various modifications to source code. Ripple effect was defined by Haney [Han72] as:

"The phenomenon by which changes to one program area have tendencies to be felt in other program areas"

It is split by Yau and Collofello into two aspects:

a) Logical: identification of program areas requiring additional maintenance to ensure consistency with an original change.

b) Performance: analysis of changes to one program area which may affect the performance of other program areas.

The technique did not provide proposals for modifying a system, but rather was applied after a number of maintenance proposals had been generated. The complexity could then be computed for each modification and the best proposal selected from both a logical and a performance perspective.

Computation of logical ripple effect involved using error flow analysis. All program variable definitions involved in an initial modification represented primary error sources from which inconsistency could propagate to other program areas. Identification of affected program areas could then be made by internally tracking each primary error source and its respective secondary error sources within the module to a point of exit. At each point of exit a determination would be made as to which error sources propagated across module boundaries. Those that did became primary error sources within the relevant modules. Propagation continued until no new secondary error sources were created. The analysis is split into two stages: lexical analysis and computation of ripple effect.

2.1.1 Code Level Logical Stability

This work was carried further in 1980 to produce a logical stability measure (the algorithm given in [YC80]). Logical stability is defined [YC80, p. 547] as:

"A measure of the resistance to the expected impact of a modification to the module on other modules within the program".

In [YC80] a software maintenance process is identified of which accounting for ripple effect is Phase 3 (Figure 1.1). Logical ripple effect is now split into two more easily comprehensible aspects: intramodule change propagation and intermodule change propagation. Intramodule change propagation was used to identify the set *Zki* of interface variables which are affected by logical ripple effect as a consequence of modification to variable definition *i* in module *k*. This requires an identification of which variables constitute the module's interfaces with other modules and a characterization of the potential intramodule change propagation among the variables inside the module. Interface variables are defined as: global variables, output parameters and variables utilized as input parameters to called modules. Intermodule change propagation is then used to compute the set *Xkj* consisting of modules involved in intermodule change propagation as a consequence of being affected by interface variable *j* of module *k*.

Xkj is calculated by first identifying all of the modules for which *j* is an input parameter or global variable. For each of these modules the intramodule change propagation emanating from *j* is then traced to the interface variables within the module. The modules involved in the intermodule change propagation as a consequence of modifying variable *i* of module *k* can then be represented by the set *Wki*. The logical complexity *LCMki* of each variable *i* in every module *k* is then computed using McCabe's Cyclomatic complexity. The probability that a particular variable definition *i* of a module *k* will be selected for modification is then estimated as:

1 / (number of variable definitions in the module)

The product of the probability with the *LCM* for each variable definition *i* gives the logical ripple effect for the module *k*, *LREk*. The logical stability measure for module *k* denoted *LSk* is the reciprocal of *LREk*.

2.1.2 Design Level Logical Stability

In the eighties the general emphasis for software measurement extended from source code measurement to measurement of design. The thinking behind this was that as design measurement gives feedback earlier in the software lifecycle, problems could be identified and eliminated or controlled before the source code was actually written, thus saving time and money.

Yau and Collofello published a paper applying the same ideas that they had used in producing their code level stability measure [YC80] to produce a design level stability measure [YC85]. The design measure analyses the module invocation hierarchy and use of global data referenced or defined in modules to produce the design stability of a program. The main difference between code level stability and design level stability is that the design stability algorithm does not consider intramodule change propagation. It produces a measure of ripple effect between modules without taking into account what happens inside them. This presupposes that information about parameters passed between modules, global variables etc... is already known. Yau and Collofello recommended that their measure be used to compare alternative programs at the design phase and to identify which portions of the program may cause problems with ripple effect during the maintenance phase. It will be seen that our approximated computation of code-level ripple effect is based on making a general assumption about intermodule flow. It might therefore be seen as sitting mid-way between Yau and Collofello's original algorithm and their proposal for a design level measure.

2.2 Ripple Effect and Software Maintenance

Software maintenance has been classified into four types [Pre94]:

- Perfective maintenance - to alter functionality
- Adaptive maintenance - to adapt software to changes in its environment
- Corrective maintenance - to correct errors

- Preventative maintenance - to update software in anticipation of future problems

Ripple effect can highlight modules with high ripple effect as possible problem modules which may be especially useful in preventative maintenance. It can show the impact in terms of increased ripple effect during perfective and adaptive maintenance where the functionality of a program is being modified or its environment has changed. During corrective maintenance it may be helpful to look at the ripple effect of the changed program and its modules before and after a change to ascertain whether the change has increased, or perhaps decreased, the stability of the program.
It is generally believed that there is a strong link between software maintenance and ripple effect. Computation of ripple effect and logical stability of a module are based on a subset of the maintenance activity: a change to a single variable definition within a module [YC80].

Regardless of the complexity of the maintenance activity being performed, it basically consists of modifications to variables within modules of code. Logical stability is computed based on the impact of these modifications. It can be used to predict the impact of primitive modifications on a program and thus be used to compute the logical stability of modules with respect to the primitive modifications. The effect of modification may not be local to the module but may affect other parts of the program; there is a ripple effect from the location of the modification to other parts of the program that are affected by the modification. If the stability of a program is poor the impact of any modification is large, hence the maintenance cost will be high and reliability may also suffer.
Several software maintenance models have been proposed in the past, Boehm's model [Boe87] consists of three major phases:

- understanding the software
- modifying the software
- revalidating the software

These are the fundamental activities of the software maintenance process. With Yau's model shown in Figure 2.1 [YC80], there is the introduction of impact analysis into the lifecycle.

The model consists of four phases, and includes analysis and monitoring of the impact of change at phase three "accounting for ripple effect". The aims of the model were to assist in achieving cost effective software maintenance and the development of easily maintainable software. Ripple effect is measured by Yau *et al* [YCM78] as part of a maintenance technique with which maintenance practitioners can understand the scope of changes made to their programs. Yau *et al* found that applying a maintenance technique based on ripple effect analysis gave benefits including: smoother implementation of changes, the injection of fewer faults, less structural degradation, a decrease in the growth of complexity and an extension to the operating life of the software.

The Pfleeger and Bohner model [PB90] has six phases, the main difference from Yau's model being that it includes "analyse software change impact" at phase two i.e. much earlier in the lifecycle. The feedback paths in the SADT model indicate attributes that must be measured; the results are then assessed by management before the next activity is undertaken. The metrics act as a controlling mechanism in the progression from existing system and change requests, to new system.

2.3 Automating Ripple Effect

Automation of ripple effect can focus in two directions: the computation of ripple effect measures; the tracing of ripple effect of variables through a program or a system.

2.3.1 Tools for Ripple Effect Measures

A prototype tool for ripple effect analysis of Pascal programs was introduced in [Hsi82]. The pseudocode algorithms used to produce the tool are presented and explained in detail. The tool consists of three subsystems: an intramodule error flow analyzer, an intermodule error flow analyzer and a logical ripple effect identification subsystem. They found that they could not identify primary error sources automatically, thus some user input was required. The intramodule error flow analyzer collects information about variable medication and use and the relationship between those medications and uses. The intermodule error flow analyzer computes summary data flow information for each module.

Then, the logical ripple effect identification subsystem traces the impact of each variable identified by the user. The tracing phase consumes a large amount of computation time, which according to S. S. Yau and S. C. Chang in [YC84] made it infeasible in some cases. Hsieh states that: "The experience we gathered using this prototype system indicates that the logical ripple effect analysis technique can be a powerful tool..." [Hsi82, p. 151]. Unfortunately no details are given of any measures produced by the tool in [Hsi82], but they are referred to for comparison by Yau and Chang in [YC84].

Yau and Chang [YC84] found that techniques for performing ripple effect analysis were taking too much computation time to be practical for large programs. They presented a new algorithm which was put forward as being much faster at computing logical stability than previous versions. Processor time for computation of logical stability for six programs was compared with the processor time using Hsieh's tool .Their algorithm does not include information from the intramodule phase as they felt that disregarding this information simplified the problem and also simulated the environment of the program design phase. Logical stability and speed of calculation for Pascal programs between 684 and 1744 lines long were compared (shown in Table 2.1).

Program	Size	#Proc.	(Hsieh) CPU sec.	(Chang) CPU sec.	Time Ratio %	Corr.
1	684	21	5312	263	5	0.94
2	1127	23	8150	119	1.5	0.81
3	487	15	1059	34	3.2	0.73
4	1744	46	44891	1318	2.9	0.68
5	1735	30	9318	551	5.9	0.58
6	1115	31	8724	54	0.6	0.16
All	**8892**	**168**	**77480**	**2339**	**3**	**0.75**

Table 2.1: Table comparing speed and accuracy of logical stability computation from [YC84]

Two algorithms are presented in [Cha84] for computing ripple effect. The first, mentioned above and implemented as a tool, does not consider intramodular information. The second, which does consider intramodular information unfortunately, is not implemented as a tool.

Yau and Chang improved the situation regarding problems with computation time but only for a limited version of the logical stability measure. Their algorithm was much faster than Hsieh's but it seems that the logical stability results only correlated with the much slower, more accurate version if the program was not very stable i.e. had a lot of global variables and ripple effect. Their approach of getting feedback at design level meant that steps could be taken to make programs more stable or highlight specific problems from an early stage. But, there is a tradeoff in that the information gained was not as accurate as information derived from code level measurement.

2.3.2 Tools for System Ripple Effect

Joiner and Tsai [JT93] used ripple effect analysis along with dependence analysis and program slicing to produce DPUTE [JT93], a Data-centered Program Understanding Tool Environment (DPUTE). DPUTE can be used during software maintenance of COBOL systems to enhance program understanding and to facilitate restructuring and

reengineering of programs. Program slicing [Wei84] is used to compute intramodule change propagation. They found that otherwise ripple effect analysis could only be semi-automatic [JT93]. This tool used a browser to highlight variables whose path can then be traced via forward or backward slicing. Different levels of ripples are shown in different colors so that users can distinguish them. The variable name, dependence type and line number in the source file are all displayed.

Problems were encountered during the automation of the intramodule change propagation stage of ripple effect analysis so a generalized program slicing technique was used to reduce the size of slices. Ripples were classified by Wang [W*et al*96] into two categories: a. Direct ripples - those introduced directly by the initial change. b. Induced ripples - those caused by direct ripple or other induced ripples.
Wang found that the average size of a potential ripple (including both direct and induced ripples) across a program could contain as much as 32% of the source code. Direct ripples affected only 1.5% of source code, thus concentrating on direct ripples was much more manageable. DPUTE only considers direct ripples.

SEMIT [CW87] is a ripple effect analysis tool which is based on both semantic and syntactic information. It creates a syntax and semantics database for software which directly links the program's semantic information with its syntax. The syntax analysis program builds an initial semantic database based on program control flow and data flow. For each procedure within a program all external data used and modified by the procedure are represented. Syntactic and semantic information is linked by grouping relations into dependencies based on "modifies-uses paths". All possible ripple effect paths are identified by SEMIT, interaction with an expert maintainer is then needed to define which the more probable paths are. The aim of SEMIT is to provide maintainers with up-to-date semantic information directly linked to the source code under observation and then express the meaning of that code, thus improving program understanding.

2.3.3 Methods for Impact Analysis and Ripple Effect in Object Oriented Programs

Before this study, there was neither an algorithm nor a tool which produces a measure for Ripple effect in Object Oriented Paradigm. All previous work in this domain was mainly concerned on studying and analyzing the impact of changes to Object-Oriented Software and then using the results for regression testing purposes.

Kung et al. [KGH94] were interested in the system-wide impact of changes for regression-testing purposes. They described a formal model for capturing and interfering on the changes to identify affected classes. The model consists of three types of diagrams: the object relation diagram (ORD), the block branch diagram (BBD), and the object state diagram (OSD). An ORD describes the inheritance, aggregation, and association relationships between the classes of a C++ library. A BBD describes the control structure and interfaces of a member function whereas; an OSD describes the state behavior of a class. Unlike modeling, these diagrams are automatically generated from the code and facilitate understanding and changing of C++ library. So in summary they defined a classification of changes (broadly, data, method, class, library), impacts resulting from the changes based on the three links inheritance, association, and aggregation, and defined formal algorithms to calculate all the impacted classes including ripple effects. However, Kung et al. did not consider the impact of data change and of method change because it had already been covered by others and also their study didn't include any tool or algorithm that measures or calculate the ripple effect.

Li and Offutt [LO96] presented algorithms to analyze the potential impacts of changes to object-oriented software, taking into account encapsulation, inheritance and polymorphism. Their technique allows software developers to perform "what if" analysis on the effect of proposed changes, and thereby choose the change that has the least influence on the rest of the system. The analysis also adds valuable information to regression testing by suggesting what classes and methods need to be retested, and to project managers who can use the results cost estimation and schedule planning. They suggested four algorithms that combine to analyze the ripple effect through the system when a component is being considered for a change. The algorithms calculate

the transitive closure of each class in "affected class set". They pick an unexpected class from the system, check all the classes that are directly related to this class via encapsulation, or inheritance, and then add all the classes that could potentially be affected by this class to the "affected class set". Li and Offutt were interested in the effects of encapsulation, inheritance, and polymorphism in their algorithms and used them in calculating the complete impact of change made in a given class. However, Li and Offutt did not consider changes in inheritance links, nor virtual methods. Also, the association and aggregation links were not fully covered in their impact calculation algorithm.

Kabaili in their paper [KKL02] "A Change Impact Model Encompassing Ripple effect and Regression Testing" have defined a change impact model for object-oriented systems. This model calculates the impacted classes due to an atomic change. They had presented an extension of the changed impact model, the first extension they took into account the classes that are impacted by ripple effect. In the second extension, they counted classes that need to be re-tested even if they are not directly affected.

Rajlich in [RAJ01] presented a prototype tool called "Ripples 2" which supports two processes of propagation of ripple effect in object oriented systems: change-and-fix and top-down propagation. However, he hasn't taken into consideration any of the object-oriented feature and their effect while counting ripple effect. He was only satisfied by checking the main program and marking the changed function and every class that uses the changed function only. However, the suggested tool "Ripples 2" doesn't count nor give a measure of the ripple effect.

Black in her paper [Bla01] describes and explains the reformation of ripple effect algorithm and its validity within the software maintenance process. She suggested an approximation algorithm using Yau and Collefello's matrix and McCabe's cyclomatic to develop Ripple effect and Stability tool (REST) which computes ripple effect for C programs.

Briand et al presented in [BLS02] a metholodgy and tool to support test selection from regression testing based on change analysis in object oriented design using UML. Their main aim was to use change impact analysis as a way for selection regression testing and for that reason they haven't extend the change impact analysis to include the object oriented dependency features such as inheritance, polymorphism and other object oriented features.

Barbara Ryder and Frank Tip [RT01] used a call graph in finding the change impact analysis. Their analysis provides a feedback on the symatic impact of a set of program changes and can be used to determine the regression test drivers that are affected by a set of changes. Moreover, if a test fails a subset of changes responsible for the failure can be identified as well as a subset of changes that can be incorporated safely without affecting any test driver.

2.3.4 Other Work

In [TM94], Turver and Munro survey existing ripple effect analysis techniques. They find that a weakness with existing ripple effect techniques is that they can not be applied in the earlier stages of the software lifecycle. To address this weakness they use Ripple Propagation Graphs (RPG) to model the hierarchical structure of system documentation with the aim of measuring the impact of a change on the entire system. A logical model of documentation is created with the relationship *consists-of* linking parts of the documentation to each other, for example, Chapter Entity *consists-of* one or more Section Entities. This information is then modeled in a RPG, and RPG crystallization performed to determine the constituent entities of the documentation [TM94, p.45]:

- Create the hierarchical graph structure of the document.
- Define a set of application themes (data objects in the documentation).
- Analyze each segment entity for themes and for each theme found, create a theme vertex and attach it to the segment vertex. This records the thematic dependencies.
- Connect together each co-occurrence of themes.

Set notation is used to represent all possible connections between documentation entities, and logical ripple effect analysis used to determine all parts of a document affected by a change. A probability connection matrix using techniques described in [Han72] and [Soo77] is also used to produce the probable maximum ripple effect. The rationale being that past recorded experience can be included in the probability matrix thus giving the maintenance manager more accurate information about their system.

Canfora [CLT96] propose a method to track the side effects of a maintenance operation to code by analyzing potential and actual relationships. Relationships existing in code are split into *potential*: exist where unit *x* may refer to any component of unit *y*, and *actual* relationships which exist where the code of unit *x* contains direct or indirect reference to some units of component *y*. Actual relationships is a subset of potential relationships and any given maintenance operation can transform a potential relationship into an actual relationship. The method is based upon the definition, use and computation of Boolean matrices. It traces ripple effect from a given point in a program and outputs a list of variables which would be affected by an initial change due to maintenance.

Several algorithms for calculating the ripple effect are presented in [YL88], they are presented as not suitable for practical use, thus are for theoretical use only. The algorithms provide ripple effect calculation for sub-sections of ripple effect analysis computation e.g. intermodule propagation.

Chapter 3

Object Oriented Dependences and Change Propagation

The new features in Object Oriented features, like encapsulation, inheritance and polymorphism make software maintenance more difficult, including identifying the parts that are affected when changes are made. Although the effects of changes in object-oriented programs can be restricted, they are also more subtle and more difficult to detect. For object-oriented systems, it is relatively easy to understand the data structures and member functions of individual classes, but the combined effect or combined functionality of the member functions is more difficult. Structured software design is typically based on functional decomposition and emphasizes control dependencies among different modules. The control dependencies among these modules are mostly hierarchical, and control dependencies only exist between the modules; hence, it is relative easy to identify the impacted modules. However, object oriented design techniques primarily use bottom up approaches. The relationships among classes form a network graph. Each class could potentially interact with any other. This makes the relationships among classes more complicated.

These complex relationships between the object classes make it difficult to anticipate and identify the ripple effects of changes. The instance of a class, the object, has its data structure, member functions (behavior), and state. The data dependencies, control dependencies, and state behavior dependencies make it difficult to define a cost-effective test and maintenance strategy to the system. An object-oriented system by implication has structure and state dependent behavior reuse, i.e., the data members, function members and state dependent behavior of a class can be re-used by another class. There are data dependencies, control dependencies, and state behavior dependencies between classes in an Object Oriented program. Polymorphism and dynamic binding imply that objects can take more than one form,

which is unknown until run time. All these features make object-oriented change impact analysis more difficult

To summarize, object-oriented systems maintenance is difficult for several reasons [KGH94]:

1) Although it is relatively easy to understand most of the data structures and member functions of the object classes, understanding of the combined effect or combined functionality of the member functions is extremely difficult.

2) The complex relationships between the object classes make it difficult to anticipate and identify the ripple effect of changes.

3) The data dependencies, control dependencies, and state behavior dependencies make it difficult to prepare test cases and generate test data to efficiently retest the impacted components.

4) Complex relations also make it difficult to define a cost-effective test strategy to retest the impacted components

In order to study and understand how to measure ripple effect in object oriented programs, we extend previous work and analyze the complex dependencies, relations and change propagations in Object Oriented code.

3. Object-Oriented Concepts

An object oriented system is composed of objects / classes. An object is composed of a set of properties, which define its state, and a set of operations, which define its behavior. The state of an object encompasses all the properties of the object plus the current values of each of these properties. Behavior is how an object acts and reacts, in terms if its state changes and message passing. The state of an object represents the cumulative results of its behavior. The constants and variables that serve as the

representation of its instance's state can be called Fields, instance Variables or Data members depend on the language. Messages result in operations that one object performs for another. Methods or Member Function are operations that clients may perform upon an object. A Class is the specification of an object; it is the "blueprint" from which an object can be created. A class describes an object's interface, the structure of its state information, and the details of its methods. Objects are runtime instances of a class. An Abstract Class is a class that only partially describes an object.

- Class A contains class B if the instance of class B is held in one of the instance variables of the A.
- Class A uses class B if A sends messages to B.
- A class can inherit the instance variables, interfaces, and instance methods of another class as if they were defined within it.
- The class from which anther class inherits is called parent of superclass.
- The class that inherits from parent is called a child, subclass or derived class. If a class has more than one parent, this kind of relationship is called multiple inheritance.
- Association is a semantically weak relationship. It cold be contains, use or inheritance.

Object-oriented software tends to encode much of the complexity in the relationships among classes, and understanding these relationships can be quite challenging. The complex relationships among classes make it difficult to anticipate and identify the ripple effects of changes. An instance of class has state (via class and instance variables) and behavior (via methods). The data dependencies, control dependencies, and state behavior dependencies make it difficult to define a cost-effective test and maintenance strategy. By implication, object-oriented software has structure and state behavior reuse, that is, the data members, function members and state dependent behavior of a class can be re-used by another class. There are data dependencies, control dependencies, and state behavior dependencies among classes in the system. Polymorphism and dynamic binding imply that object references can

refer to objects of different types, and which type is not known until execution. All these features make object-oriented maintenance more difficult.

3.1 Change Impact Definitions

In structured programming, one thinks in terms of inputs, functions and outputs. In object oriented programming (OOP), the approach is different -- a message is typically passed to an object to request an operation on the object. Objects have *methods* and *data members*; the methods specify the operations allowed on the object's private data, and the data members specify the state information for the object. Henceforth, we refer to either a method or data member as *class member*. When a class member changes, it might impact other classes through message passing, inheritance, etc...

The basic component in our analysis is the class. A class is composed of member functions and member variables.

Direct Relationship: There is a direct relationship R between class A and B (A*R*B) if A and B have one of the three kinds of relationships: *Containment*, *Use*, or *Inheritance*. They are defined by [LI98] as follows:

Containment: Class A *contains* class B if B declared as a class member of A.

Use/Reference: There are several ways that a use/reference relationship can be formed.

- **Containment:** If class A *contains* class B, then class A *uses* Class B. If A contains B by reference that means that A contains a reference to B. B's life span can be longer than A's.

- **Classes passed in as method parameter:** If a method m of class A takes parameters P1 ... Pn, we say class A *uses* each pi, i = 1...n, and m is in the reference sets of each of Pi. Pi can be any class and type

- **Classes referenced in the left hand side of assignment:** If class A or one of its members is specified in the left hand side of the assignment statement, A or its member is *defined* by all the variables on the right hand side. Thus, class A (or its member) belongs to the reference set of all those variables on the right hand side of the equation.

- **Return type of method:** The return type of a method m is defined by m. m belongs to the reference set of this return type. Since the parameters may not be used in the body, and their effect may not direct impact the return type, we do not consider the return type to be defined by these parameter types. If the return type is defined by a parameter, it will show up in the analysis of this method body.

- **Variables declared in a method:** Any variable that is referenced in the method m can be considered to be used by m and can be put into the reference set of m

Inheritance relationship: Class A inherits from Class B if B is declared as a super class of A.

3.2 Object-Oriented system dependences and type of changes

A *dependency* in a software system is a direct relationship between classes X and Y entities in the system X R Y such that a modification to X may affect Y [WH92].

Wilde and Huitt [WH92] classified dependencies as: (1) data dependencies between two variables, (2) calling dependencies between two modules, (3) functional dependencies between a module and the variables it computes, and (4) definitional dependencies between a variable and its type. We present OO dependences in more details as follows and classify them into five types of changes that may occur in Object Oriented code.

i) Class-to-Class Dependencies:

a) C1 is a direct super class of C2 (C2 inherits from C1)

b) C1 is a direct sub class of C2 (C1 inherits from C2)

c) C1 is an ancestor class of C2 (C2 indirectly inherits from C1)

d) C1 uses C2 (C1 references C2, include direct reference and indirect reference)

e) C1 contains C2

- C1 contains C2 by value
- C1 contains C2 by reference

ii) Class to Method:

a) Method M returns object of Class C

b) C implements method M

iii) Class to Variable:

a) V is an instance of Class C

b) V is a class variable of C

c) V is an instance variable of C

d) V is defined by class C

iv) Method to Variable:

a) V is a parameter for method M

b) V is a local variable in method M

c) V is imported by M (i.e. is a non-local variable used in M)

d) V is defined by M

v) Method to Method:

a) Method M1 invokes method M2

b) Method M1 overrides M2

3.3 Types of Changes and their Relationship

The typical changes that may be made to Object Oriented programs, from a syntactic point of view, can be listed as follows:

i) System level change:

a) Add super class

b) Delete super class

c) Add sub class

d) Delete sub class

e) Delete an object pointer

f) Delete an object reference

g) Add an aggregated class

h) Delete an aggregated class

i) Change inheritance type

- Change from public inheritance to private inheritance
- Change from private inheritance to public inheritance

ii) Class level change:

a) Add member

b) Delete member

c) Define/Redefine member

d) Change member

I. Change member access scope:

1) Change from public to private

2) Change from public to protected

3) Change from protected to public

4) Change from protected to private

5) Change from private to public

6) Change from private to protected

II. Change method:

1) Protocol change:

a) name change

b) parameter change

c) return type change

III. Change Data member:

1) Add data declarations

a) Delete data declarations

b) Add data definitions

c) Delete data definitions

d) Change data declaration

- Change data type
- Change data name

(e) Change data definition

IV. Function implementation change

e) Add/delete an external data use

f) Add/delete an external data update

g) Add/delete/change a method call

h) Add/delete a sequential segment

i) Add/delete/change a branch/loop

j) Change a control sequence

k) Add/delete/change local data

l) Change a sequence segment

Inheritance is assigned the greatest impact power, with containment relationship as medium and use relationship the least, because we think the impact power of inheritance is greater than the impact power of containment, and it is greater than the impact power of use. Inheritance is considered to have the highest impact power because super-classes define subclasses behavior. Any changes in the public and protected levels of the super class will impact its sub classes. A containment relationship implies the use relationship with additional constraints, like the life span of the contained object may be the same as that of the container class. The contained class constructors and destructors are always called by the container class. So the impact power of containment is considered to be greater than that of the use relationship. Since the coupling between inheritance is much higher, its impact power is assigned a higher value than that of containment and use.

3.4 Object Oriented Features and Relationships:

Although objects are more easily identified and packaged, however, features such as encapsulation, inheritance, aggregation, polymorphism and dynamic binding can make the ripple effects in Object Oriented programs more difficult to understand and handle than structured programs and will make the maintenance process more complicated. Therefore, I have explored each object-oriented feature listed above and its relations with ripple effect

3.4.1 Encapsulation

Encapsulation provides an effective way to enforce information hiding because the data aspect of an object may made private and access to these private data can be achieved only through operations of the object i.e. through the methods defined within the object. Therefore, this way of implementation within one class will have no ripple effect on other classes and on the contrary the effect of change can be minimized.

Note: in the presence of encapsulation, the only way to observe the state of an object is through its public methods.

3.4.2 Inheritance

It's the process by which one class can acquire the properties of another class. Therefore, a new class can be defined in the object oriented paradigm without starting from the beginning. As a result, super class can be reused by the subclass. Methods inherited from a super class must be retested in the context of the subclass because a change in one of the method will have a ripple effect in the subclass. Derived class reuses both data members and the function member of a base class and therefore, a change in the data/function member will make a ripple effect to the derived class.

3.4.3 Polymorphism

It's the ability to take more than one form. An attribute may have more than one set of values and an operation may be implemented by more than one method. One problem of polymorphism is that we couldn't know which method or value will be used before runtime.

3.4.4 Dynamic Binding

Is the method that implement an operation is unknown until runtime. As a result, data binding feature makes the ripple effect measurement more difficult.

3.5 Change Propagation

Change propagation in Object Oriented programs can be classified into intra-class propagation and inter-class propagation.

3.5.1 Intra Class change propagation

Is the propagation inside a method's class: data element inside a method/function inside the class:

1) Local Variables:

a) The variable is defined in an assignment statement.
b) The variable is assigned a value read from an input.
c) The variable is an input parameter to module m.
d) The variable is an output parameter from a called module m.

2) Global / or inherited variables:

a) A variable with modifier "private" is global to all the methods in the class but not visible outside the class.
b) Inherited variable.

3) Global or inherited methods.

4) Inherited methods/variables (may be included by global variables/functions)

3.5.2 Inter-class change propagation

Data dependency/relationship among different methods and functions inside and outside the class.

1) Global variables.
2) Global methods.
3) Public/ protected members of a super class in case of inheritance.
4) All Public members of any class inside the system
5) Variable is an input parameter to a called method/class (via a message).
6) Variable is an output parameter of a method.

3.5.3 System dependency propagation

Classes interaction with each other.

3.6 Metrics and the Object Oriented Ripple Effect:

Keremer and Chidamber introduced complexity metrics for object-oriented systems in their paper [CK91] and because the ripple effect is a measure of complexity, I had studied these metrics and classified them into inter-class and intra-class metrics after I added some other object-oriented complexity metrics. I had explored each metric alone and its relationship and effect on ripple effect

3.6.1 Inter-Class Metrics:

1. **Depth of Inheritance tree (DIT):** The depth of a class within the inheritance hierarchy is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. The higher the DIT of a class, the more ancestor classes it has. Deeper trees require greater design complexity since more methods and classes are involved. A subclass does not only depend upon its direct super class, but also upon its ancestor classes as it inherits their features. This increases the ripple effect of subclasses because changes to their ancestors may necessitate changing them.

2. **Number of children (NOC):** NOC is the number of direct descendants (subclasses) for each class. Classes with large number of children are considered to be difficult to modify and usually require more testing because of the effect on changes on all the children. The more subclasses a class has, the higher its incoming dependencies since these subclasses depend upon it. Since only the outgoing dependencies of a class affect its stability and thus having a greater ripple effect, NOC metric is not expected to have correlation with class stability nor ripple effect.

3. **Coupling between objects (CBO):** CBO is defined as the count of the classes to which this class is coupled. Coupling is defined as: Two classes are coupled when methods declared in one class use methods or instance variables of other class. High CBO of a class means that this class depends upon many classes (outgoing dependencies) and/or many classes depend upon it (incoming dependencies). The outgoing dependencies of a class reduce its stability since they represent external influence to make it change and as a result increase the ripple effect.

4. **Response for a class (RFC):** RFC is defined as number of methods in the set of all methods that can be invoked in response to a message sent to a message sent to an object of a class. The higher the RFC of a class, the higher the number of internal and external methods available to this class. The external methods make this class depends upon the class in which these methods are defined. This in turn may require modifying this class whenever these external methods are modified. So this decrease class stability and increases class ripple effect.

5. **Number of children in a sub tree (NOC-ST):** When some component of a class is changed, it may affect not only its children but the whole sub-tree of which the changed class is the root.

6. **CBO No Ancestors CBO-NA**: It is same as CBO but the coupling between the target class and its ancestors is not taken into consideration The coupling between the target class and its ancestors, taken into consideration by CBO, is irrelevant for change impact, since the ancestors of the target class will never be impacted. To eliminate such "noise", ancestors are excluded in CBO_NA.

7. **CBO-IUB:** (CBO is Used By: the part of CBO that consists of the classes using the target class): (the part of CBO that consists of the classes using the target class) the definition of CBO merges two coupling direction: classes using the target class and classes used by the target class. Very strong correlated with change impact good indicator for changeability in system.

8. **CBO-U** (CBO Using: the part of CBO that consists if the classes used by the target class): introduced as a consequence of CBO_IUB, to cover the part of CBO not considered by CBO_IUB.

3.6.2 Intra-Class Metrics:

1. **Weighted Methods per class (WMC):** WMC measures the complexity of an individual class. The larger the number of methods in a class, the greater the potential impact on subclasses, since these subclasses will inherit all the methods defined in a class. High WMC of a class suggests that this class has many methods and/or its methods have high complexity. This increases the likelihood of having some methods that use methods and/or instance variables of other classes. If so, this makes this class depends upon other classes and thus increase its ripple effect (reduces its stability).

2. **Lack of cohesion in method (LCOM):** LCOM is defined as the number of different methods within a class that reference a given instance variable. High LCOM of a class suggests that this class is weekly cohesive and does not promote encapsulation. This increases the likelihood that this class be dependent upon other classes, which increases the ripple effect.

The more a class is used through invocation of its methods and outside reference to its variable, the larger the impact of change to such a class and therefore, the higher the ripple effect.

Chapter 4

Counting Ripple Effect as a Matrix Product

The purpose of this chapter is to give a precise definition of the calculation of ripple effect and logical stability of object-oriented programs. The computation of ripple effects for object-oriented programs is done in 2 levels: code level and architecture level. Two fundamental ideas in the computation of ripple effect at the code level are: intra-class and inter-class propagation. Whereas, in the architecture level, we calculate the interactions between the classes with each other at the system level. This chapter gives a detailed description of what they are and the way in which they are calculated. This is followed by the computation of ripple effect for an example program to clarify the use of the ripple effect algorithm in practice.

4.1 Intra-class Change Propagation

The computation of ripple effect is based on the effect that a change to a variable will have on the rest of a program. Consider the lines of code contained in C, shown in Figure 4.1. A change to the value of d in line 1 will affect the value of a in line 1, which will propagate to a in line 2. In line 2 a will affect d which will then propagate to d in line 3. Propagation of change from one line of code to another within a class is called *intra-class change Propagation.* Clearly, propagation takes place from definitions to uses of variables and via assignments.

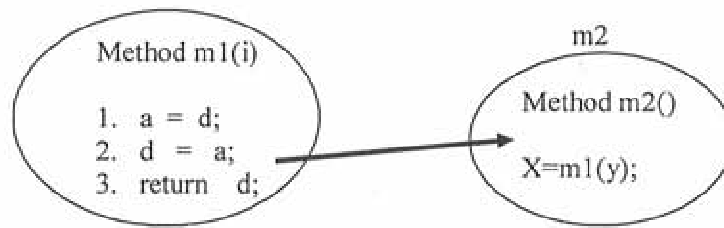


Figure 4.1: Example code of class C

```
Class C {
        int a;
        Public int m1() {
                int d;
                a = d;  // line 1
                d = a; // line 2
                return d; // line 3
        }

        Public void m2() {
                int y;
                .
                .
                int x = m1(y);
                z = a;
        }

        Public void m3() {
                int f;
                .
                .
                f=a;
        }
}
```

example 4.1

Starting points for intraclass change propagation such as *a* in line (1) can be thought of as "definitions", the variable is being defined or given a value. Propagation then emanates from the defined variable through the class across the class boundary and into other classes (inter-class change propagation).

Intra-class ripple computation due to a change in a variable is based on the following five conditions: (refer to figure 5.1 example.java)

1. The variable is defined in an assignment statement. For example '*caverage*' in

```
30      caverage = total/ccounter;
```

2. The variable is assigned a value which is read as input. For example '*s*' in

```
17    s = in.readLine();
```

3. The variable is an input parameter to method *m*. For example 'ccounter' in

```
28 Private float calcmean (float ccounter)
```

4. The variable takes returned value from a called method. For example *counter*:

```
7       Counter = values();
```

5. The variable is a global or inherited variable. For example '*total*' in

```
30      caverage = total/ccounter;
```

In calculation of intra-class propagation, if a variable is global to the method, to determine whether it qualifies as a starting point for matrix V_m will depend on the visibility rule of the particular object-oriented language used. For example, if a variable with a modifier private is "global" to all methods in the class but not visible outside the class. Therefore, the ripple computation engine can assume that if java code is passed we can treat all variables no defined locally in a method as being global to the method. The computation engine does not need to be concerned about the underlying visibility rules or the data hiding of variables within each class. In most statically complied procedure languages, it can before run-time which piece of code will be entered after the invocation of a function. A characteristics of all object-

oriented languages however, is that the binding of at least some calls to a particular function code only takes place at runtime. Different object-oriented languages provide different mixes of static and dynamic binding. In Java for example, methods are virtual by default, which means that calls to all methods will be bounded at runtime.
In addition, in Java where a method is not abstract or part of an interface and is not overridden by a method with the same signature in a subclass which implies that we can able to determine before runtime the method code executed when the method is invoked. As a result, the ripple effect computation engine can determine such methods by methods by examining the source code class specifications of the system.

Intuitively only global values on the right hand side of assignments should count. Any variable occurrence on the left hand side is receiving a value from the variable on the right hand side of the assignment, thus whether it is global or not is irrelevant. In this instance if *average* is global this specific occurrence is not going to affect anything. A 0-1 vector V_m is used to represent the variable definitions in method *m*. Variable occurrences that satisfy any of the above conditions are denoted by "1" and those which do not by "0". We shall use the notation x_i^d (x_i^u) to denote a *definition* (*use*) of variable *x* at line *i*. For example, *a* means variable *a* is *defined* in line *1* and *a* means variable *a* is *used* in line *2*. Vector V_m for the code in our example in Figure 4.1 (where a is assumed global) is therefore:

$$\begin{array}{lccccccc} & & a_1^d & d_1^u & d_2^d & a_2^u & d_3^u & \\ V_{m1} = & (& 1 & 0 & 1 & 1 & 0 &) \end{array}$$

A 0-1 propagation matrix *Pm* can be produced to show which variable values may propagate to other variables within method *m*. The rows and columns of *Pm* represent each individual occurrence of a variable. Propagation is shown from row *i* to column *j*. For example, the propagation from *a* in line 2 to *d* in line 2 is shown at row 4 column 3 and not at row 3 column 4.That is P is not symmetric. For the code of example 4.1 we get the following matrix:

$$
P_{m1} = \begin{array}{c} \\ a_1^d \\ d_1^u \\ d_2^d \\ a_2^u \\ d_3^u \end{array}
\begin{array}{c}
\begin{array}{ccccc} a_1^d & d_1^u & d_2^d & a_2^u & d_3^u \end{array} \\
\left(\begin{array}{ccccc}
1 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1
\end{array} \right)
\end{array}
$$

We observe that *Pm*1 is reflexive and transitive; that is, every variable occurrence is assumed to propagate to itself, and if *v*1 propagates to *v*2 and *v*2 propagates to *v*3 then *v*1 also propagates to *v*3. *Pm* therefore represents the transitive closure of variables within module *m*. In graph theory terms we conclude that *Pm*1 represents the reachability matrix of a directed graph [PY76]; that is, it shows how element (i.e variable/line) can reach any other element in the code. The nodes of the graphs are the variable occurrence represented in the vector V and the edges are given by the direct-change-propagation relations: (a) change propagates either from the right-hand side of an assignment to the left-hand side, or (b) it propagates from the definition of a variable to a subsequent use of that same variable. This directed graph can be represented by an adjacency matrix R. The propagation (reachability) matrix is then given by the union of I, R, R^2, … R^n, where n is the number of variable occurrences.

4.1.1 Decomposition of Matrix *Pm*

Matrix *Pm* represents intra-class change propagation. This is clearly a transitive relation because if change propagates from variable occurrence v_i to v_j , also from v_j to v_k, then a change in occurrence *vi* will propagate to v_k via v_j . Thus, as we observed earlier, *Pm* is the matrix of a transitive relation and represents the reachability matrix of some basic relation *Bm*. To determine *Bm* is not difficult: change propagates either from the right-hand side of an assignment to the left-hand side, or it propagates from

the definition of a variable to a subsequent use of that same variable. These two modes of propagation are referred to as 'assignment' and 'definition/use', respectively. If we treat the two as different relations, represented by matrices *Am* and *Dm*, respectively, then we see that $Bm = Am + Dm$.

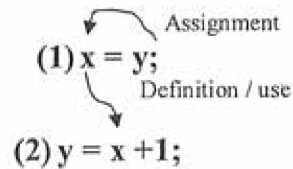


Figure 4.2: Assignment and Definition/ use

To explain this we turn to Figure 4.2 where information flow from one variable occurrence to another is shown using arrows, variable occurrence *x* takes its value from *y* in line 1, thus *x,y* is an assignment pair. Information about such pairings is held in matrix *Am*. The definition of *x* in line 1 is used by *x* in line 2. This is a definition/use association. Information about definition/use associations is held in matrix *Dm*.

The combination of information from assignment and definition/use gives us information about the flow of values from one variable to another within a class. From this information we can work out which variables would be affected if we changed any particular variable occurrence. The assignment matrix *Am* which holds information about all assignment pairings for our example code is as follows:

$$
A_m = \begin{array}{c} \\ x_1^d \\ y_1^u \\ y_2^d \\ x_2^u \end{array}
\begin{array}{cccc} x_1^d & y_1^u & y_2^d & x_2^u \end{array}
\left(\begin{array}{cccc}
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0
\end{array} \right)
$$

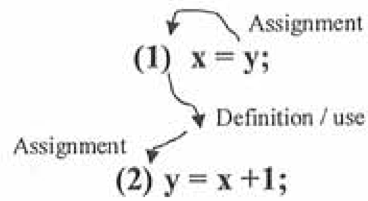


Figure 4.3: Assignment and Definition/ use information held in Matrix B

The definition/use association matrix *Dm* is as follows:

$$
D_m = \begin{array}{c} \\ x_1^a \\ y_1^u \\ y_2^a \\ x_2^u \end{array}
\begin{array}{cccc} x_1^d & y_1^u & y_2^d & x_2^u \end{array}
\left(\begin{array}{cccc}
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{array} \right)
$$

Matrix *Am* and matrix *Dm* have all variable occurrences as rows and columns, even though in *Am* only defined variables are needed as columns and in *Dm* only defined variables are needed as rows. The sum of these matrices then gives us matrix *Bm* representing direct intra-class change propagation. The information now held in matrix *Bm* is also shown in Figure 4.3.

$$B_m = \begin{array}{c|cccc} & x_1^d & y_1^u & y_2^d & x_2^u \\ \hline x_1^d & 0 & 0 & 0 & 1 \\ y_1^u & 1 & 0 & 0 & 0 \\ y_2^d & 0 & 0 & 0 & 0 \\ x_2^u & 0 & 0 & 1 & 0 \end{array}$$

We can now find the reachability matrix (equivalent to the transitive closure) for B_m, namely P_m, using:

$$Pm = I \vee B \vee B^2 \vee \ldots \vee B^n$$

n = number of variable occurrences, in this case four.

The reachability matrix shows all possible links between any variable occurrence and any other variable occurrence within the module. From the information now contained within matrix Z any change to a variable occurrence can be tracked throughout the class and the ramifications of its change calculated.

$$
P_m = \begin{array}{c} \\ x_1^d \\ y_1^u \\ y_2^d \\ x_2^u \end{array}
\begin{array}{c} \begin{array}{cccc} x_1^d & y_1^u & y_2^d & x_2^u \end{array} \\
\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}
$$

4.1.2 Problems in calculating inter-class propagation

When tracing back to where the object was originally created, it is sometimes possible to determine before runtime the actual method executed. However, in some cases, it is so difficult and sometimes impossible.

Example:

Where a super type variable is passed as a parameter to a method suppose, for example, that classes A1 and A2 implement interface IA which specifies the signature of method method1.

In the method bellow:

```
Void someMethod (IA anIA) {
        anIA.method1();
}
```

It is not possible to determine which version of method1() will actually be executed. this depends on the class of the object to which anIA refers. However, for the purpose of ripple effect calculation, the code could be re-written as follows:

```
void sometod (IA anIA) {
        if (anIA instancof A1)
          ((A1)anIA).method1();
        else
          ((A2)anIA).method2(); }
```

4.2 Inter-class change propagation

Propagation across classes from one method to another is called *inter-class change propagation*. A change to a variable can propagate to other method if: (refer to figure 5.1 example.java)

1. The variable is an inherited or global variable. For example *total*:

```
30      caverage = total/ccounter;
```

Intuitively only globals on the left hand side of an assignment should count - this specific occurrence of *total* is not propagating to anywhere, but we are sticking to Yau and Collofello's definition.

2. The variable is an input parameter in an inter-class message. For example *counter*:

```
8       Average = object.calcmean(counter);
```

3. The variable or object is returned by a method *m* to a method in another class. For example *vcounter*:

```
26      return(vcounter);
```

For example, in the code of method m1 in Figure 4.1, *d* clearly propagates to any method calling *m*1. If *a* is global/inherited then its occurrence on the left-hand-side of the assignment in line '1' will cause propagation to any method using *a*. Suppose that the code constituting method *m*1 is called by another method *m*2, that *a* is inherited/global and method *m*2 uses *a*, and that yet another method *m*3 uses *a* and *d*. The (i,j)th entry is 1 if variable *i* propagates to method *j*. We can represent the propagation of these variables using a 0-1 matrix *Xm*1:

$$V_{m1}P_{m1}X_{m1} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 3 \end{pmatrix}$$

In this example, we can infer from the results of the matrix A = $Vm1Pm1Xm1$ that there are 0 propagations from method *m1* to method *m*1, 1 to method *m*2 and 3 to *m*3.

4.3 Complexity measure logical stability and ripple effect

A complexity measure is factored into the computation by Yau and Collofello so that the complexity of modification of a variable definition is taken into account. Matrix C represents McCabe's cyclomatic complexity [McC76] for the methods in our code, shown in example 4.1 (the values for *m*2 and *m*3 have been chosen at random) (We can also use the Average Method Complexity for object oriented programs which is:

$$AMC = \frac{1}{n}\sum_{i=1}^{n} Ci$$

Where C1...Cn are the static complexity of methods we can measure the static complexity using a static complexity metric such as McCabe's cyclomatic complexity metric)

The matrix A of Section 4.2 yields a count of the amount of propagation that ripples across classes. However, the effect of change is not identical in all classes. To capture the differences, we propose factoring in a complexity measure for each class in order to give a more reflective measure of the ripple effect. This complexity measure can be based on the metrics discussed in chapter 3.

We propose using a weighted sum of these metrics for each class/method as follows:

$$(a.HIT + b.RFC + c.CBO + d.WMC + e.LCOM) / 5$$

where the coefficients a-e are normalization coefficients that aim to yield comparable dimensions for the five metrics. A simple way to determine these coefficients can be to make them all equal to the reciprocal of the average of the total sum of all five metrics over all classes in an OO program.

Assuming that the complexity values for the example used in Section 4.2 are computed, we obtain a complexity vector, C (for the three classes):

$$C = \begin{matrix} m_1 \\ m_2 \\ m_3 \end{matrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

The product of $Vm1Zm1Xm1$ and C is:

$$Vm1.Zm1.Xm1.C = (0 \ 1 \ 3) \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = 4$$

This number represents the complexity-weighted total variable-change propagation for method $m1$. We normalize this number by dividing it by the number of variable definitions in method $m1$ ($1/|Vm1|$) to give the mean complexity-weighted variable-change propagation per variable definition in method $m1$. In our example, the number of variable definitions in m1 is 3, $|Vm| = 3$, (equals number of 1s in $Vm1$). Therefore, ripple effect for method $m1$ is defined as:

$$RE = (Vm1.Pm1.Xm1.C)/3 = 4/3 = 1.33$$

The logical stability measure for method $m1$ is defined to be the reciprocal of the ripple effect value. In our example, LS = 3/4 = 0.75. These numbers for the ripple

effect and logical stability should be read as relative and not absolute. That is, they can be used to compare RE and LS values of different methods in different classes in the same program with each other.

Hence that no class with a Ripple Effect of zero was found in this study, but if there were we could adopt a definition of Logical Stability being $1/(1+RE)$.

4.4 Architecture Level

The computation of the ripple effect at the architectural level is based on the effect that a change to a single class will have on the rest of the program. A high-level change can be an internal code change in the class, but is considered as an overall class change. Alternatively, it can be a change in one of the relations if this class with other classes in the program. That is, for the architectural level, we do not consider the data flow level. A 0-1 vector Vc can be used to represent the classes definitions in a program. In this vector, a class in the program that has a direct effect on other classes is denoted by "1" otherwise it is denoted by "0". A direct effect includes inheritance, aggregation or a directed association relation (i.e sends value). For example, assume a program with the following architectural characteristics:

Class 1; Class 2 extends class 1; Class 3 extends class 1; Class 4 extends class 3;

Class 5 extends Class 2; then,

	C1	C2	C3	C4	C5	
Vc = (	0	1	1	1	0	)

Also, a class relation matrix Rc can be determined from the class graph diagram such that classes are the nodes of the graph and the directed relations define the edges.

A 0-1 propagation matrix Pc can be produced to show which classes will propagate to other classes within the program. Pc is computed from Rc in a similar way to that described in chapter 4 for Pm. Propagation is shown from row i to column j; for example:

$$Pc = \begin{array}{c} \\ C1 \\ C2 \\ C3 \\ C4 \\ C5 \end{array} \begin{array}{c} \begin{array}{ccccc} C1 & C2 & C3 & C4 & C5 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

The amount of propagation is given by the matrix product Ac=VcPc. For our example,

$$Vc\,Pc = (1\,1\,1\,1\,0) \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = (1\ 2\ 3\ 2\ 1)$$

Now we multiply it by the complexity measure. In object Oriented the Weighted Method per class (WMC) is used to measure the complexity of a class. The WMC is calculated as the sum of McCabe's cyclomatic complexity (or Halstead Complexity) of each local method. Assume that the vector class complexity measures, C, for our example is

$$C = \begin{array}{c} C1 \\ C2 \\ C3 \\ C4 \\ C5 \end{array} \begin{pmatrix} 3 \\ 2 \\ 2 \\ 1 \\ 1 \end{pmatrix}$$

$$VcPcC = \begin{pmatrix} 1 & 2 & 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = 16$$

This number represents the complexity-weighted total class-change propagation for a program. We now multiply this by the reciprocal of the number of class definitions in the program (= number of 1s in Vc) to get the mean complexity-weighted class-change propagation per class. That is, the ripple effect for program *P* is defined to: RE=16/4 = 4.Further, the logical stability measure for Program *P* is defined to be its reciprocal: 4/16 = 0.25

Chapter 5

Examples

It is desirable to automate the computation of ripple effect measure as the process is complex and tedious by hand. But even when automated, computation of ripple effect can be time consuming. Yau and Chang [YC84] give an example of a 2000 line program's stability measure taking thirteen hour of CPU time to compute. Problems were encountered during the automation of intra-class change propagation stage of ripple effect analysis for which program slicing was used, ripple effect could otherwise only be computed semi-automatically. In this chapter I described in details how to compute the ripple effect and logical stability to 2 classes: example.java and calculations.java using the techniques described in section 4.

5.1 Computing Ripple Effect for an Example Class

The following is an example of the computation of logical stability of a program called *Example.java*. A listing of the program source code is given in Figure 5. The following algorithm can be used to calculate ripple effect and logical stability. It is split into eleven separate steps; steps one to nine should be followed iteratively for each method, steps ten and eleven for the class as a whole.

```
1: import java.io.*;
2: import java.awt.*;
3: Public Class example
4: Float total ;
5:  Public Void main() {
6:    Float average, power, counter;
7:    Counter = values();
8:    Average = calcmean(counter);
9:    Power = calpower ();
10:   Output(average,power);
11: }

12: Private float values() {
13:   float vnumber;
14:   float vcounter = 0;
15:   String s = new String();
16:   System.out.println("Enter a value or -1 to calucuale mean:");
17:   s = in.readLine();
18:   vnumber = Float.valueOf(s).floatValue();
19:     for (; vnumber!=-1; vcounter=vcounter+1)
20:   {
21:     total = total + vnumber;
22:     System.out.println("Enter a value or -1 to calucuale mean:");
23:     s = in.readLine();
24:     vnumber = Float.valueOf(s).floatValue();
25:   }
26:   return(vcounter);
27: }
28: Private float calcmean (float ccounter) {
29:   float caverage;
30:   caverage = total/ccounter;
31:   return(caverage);
32: }
33: Private float calcpower() {
34:   float xpower;
35:   xpower = pow(total,3);
36:   return(xpower);
37: }
38: Private void output (float oaverage, float opower){
39:   System.out.println("A = total of values:",total);
40:   System.out.println("B = mean of values:",oaverage);
41:   System.out.println("C = A cubed:",opowertotal);
42: }
43:}
```

Figure 5.1 Example.java

STEP 1: Make a list of all occurrences of variables (not including variable declarations) for each method in the order in which they appear in the source code. For example, in method *values* the list would be:

vcounter, vnumber, vnumber, vcounter, vcounter, total, total, vnumber, vnumber, vcounter

STEP 2: Form Boolean vectors for each method. We give *Vvalues* as an example, with '1' in the vector denoting that the variable occurrence satisfies one or more of the five conditions specified in section 4.1, and '0' that it satisfies none. By inspection of the source in example.java

	vc_{14}^{d}	vn_{18}^{d}	vn_{19}^{u}	vc_{19}^{d}	vc_{19}^{u}	tot_{21}^{d}	tot_{21}^{u}	vn_{21}^{u}	vn_{24}^{d}	vc_{26}^{u}
V_{values} = (	1	1	0	1	0	1	1	0	1	0)

STEP 3: Construct matrix *Zm* for each method:

	vc_{14}^{d}	vn_{18}^{d}	vn_{19}^{u}	vc_{19}^{d}	vc_{19}^{u}	tot_{21}^{d}	tot_{21}^{u}	vn_{21}^{u}	vn_{24}^{d}	vc_{26}^{u}
vc_{14}^{d}	1	0	0	1	1	0	0	0	0	1
vn_{18}^{d}	0	1	1	0	0	1	1	1	0	0
vn_{19}^{u}	0	0	1	0	0	0	0	0	0	0
vc_{19}^{d}	0	0	0	1	1	0	0	0	0	1
vc_{19}^{u}	0	0	0	1	1	0	0	0	0	1
tot_{21}^{d}	0	0	0	0	0	1	1	0	0	0
tot_{21}^{u}	0	0	0	0	0	1	1	0	0	0
vn_{21}^{u}	0	0	0	0	0	1	1	1	0	0
vn_{24}^{d}	0	0	1	0	0	1	1	1	1	0
vc_{26}^{u}	0	0	0	0	0	0	0	0	0	1

STEP 4 : Construct matrix *Xvalues*:

	Main	values	calcmean	calcpower	output
vn^{d}_{14}	0	0	0	0	0
vn^{d}_{18}	0	0	0	0	0
vn^{u}_{19}	0	0	0	0	0
vc^{d}_{19}	0	0	0	0	0
vc^{u}_{19}	0	0	0	0	0
tot^{d}_{21}	0	0	1	1	1
tot^{u}_{21}	0	0	0	0	0
vn^{u}_{21}	0	0	0	0	0
vn^{d}_{24}	0	0	0	0	0
vc^{u}_{26}	1	0	0	0	0

STEP 5: Construct matrix *C* representing the McCabe complexities for the methods in this class which are as follows:

C =

main	1
values	2
calcmean	1
calcpower	1
output	1

STEP 6: Find the Boolean product of the matrices formed in *STEP 3* and *STEP 4* *ZvaluesXvalues*:

$$\begin{pmatrix} 1&0&0&1&1&0&0&0&0&1 \\ 0&1&1&0&0&1&1&1&0&0 \\ 0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&1&1&0&0&0&0&1 \\ 0&0&0&1&1&0&0&0&0&1 \\ 0&0&0&0&0&1&1&0&0&0 \\ 0&0&0&0&0&1&1&0&0&0 \\ 0&0&0&0&0&1&1&1&0&0 \\ 0&0&1&0&0&1&1&1&1&0 \\ 0&0&0&0&0&0&0&0&0&1 \end{pmatrix} \begin{pmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&1&1 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \end{pmatrix} = \begin{pmatrix} 1&0&0&0&0 \\ 0&0&1&1&1 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \\ 1&0&0&0&0 \\ 0&0&1&1&1 \\ 0&0&1&1&1 \\ 0&0&1&1&1 \\ 1&0&0&0&0 \end{pmatrix}$$

STEP 7: Find the product of the matrices formed in *STEP 2* and *STEP 6*, *VvaluesZvaluesXvalues*:

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 4 & 4 & 4 \end{pmatrix}$$

STEP 8: Find the product of the matrices formed in *STEP 2* and *STEP 5*, *VvaluesZvaluesXvalues*:

$$\begin{pmatrix} 2 & 0 & 4 & 4 & 4 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = 14$$

STEP 9: Divide this figure by $|Vmj|$, the number of variable definitions in *class C* to give the measure of ripple effect for the module. Looking at *Vvalues* we can see that there are 6 '1's in the vector, we therefore divide the product of the matrices by 6 giving:

14 / 6 = 2.33

The Ripple Effect values for the other methods are:

REmain = 1.33

REcalcmean = 1

REcalcpower = 1

REoutput = 0

The Logical Stability for method *values* is therefore 6 / 14 = 0.43 and for the other methods in class example.java:

LSmain = 0.75

LScalcmean = 1

LScalcpower = 1

LSoutput = 0

STEP 10: Calculate the Ripple Effect for the class as a whole using:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{Vmi.Zmi.Xmi.C}{|Vmi|}$$

Where m = method and n = number of methods in class.

Therefore *REPexample:java* = 34/30 = 1.13

STEP 11: Finally the Logical Stability measure for the Class *Example.java* is the reciprocal ripple effect measure:

LSP = 1/*REP* = 30/34 = 0.88

The final results are shown in table 5.1.

Class example.java	*Line of Code*	*Ripple Effect*	*Logical Stability*
Main	12	1.33	0.75
Values	5	2.33	0.43
Calcmean	3	1	1
Calcpower	3	1	1
Output	3	0	0
Class Results	**26**	**1.13**	**0.88**

Table 5.1 REA results

Table 5.1 shows the impact in term of increased ripple effect during perfective and adaptive maintenance where the functionality of a program is being modified or its environment has changed. If the stability is poor, the impact of any modification is large and this means the maintenance cost will be high and reliability will suffer.

The ripple effect shows us how the methods/classes are stable. In our example, the method values have the highest ripple effect 2.33 and this means that this method is the least stable in class example with logical stability 0.43. The method output has zero ripple effect and as a result it has zero logical stability. This means that whatever we change in class example.java the method output is always stable and it not going to affect other methods inside the class. As ripple effect increases the logical stability of this program decreases which means that methods with higher ripple effect will face more problem when being updated. In our example, updating the method Values will have an impact on methods calcmean and calcpower and as a result the logical stability of the whole class will be decrease.

The matrix we used in calculating the ripple effect and logical stability are useful and can describe each part inside the class which make us understand it better. Matrix Z represents the variables' propagations to each other inside a certain method for

example, vnumber propagate to total in line 21. On the other hand, Matrix X represents the variables propagation to another method within the same class for example, variable total propagate to methods calcmean and calpower Inter-class propagation of all variable occurrences inside a class can be found by finding the Boolean product of Z*X. Whereas, V*Z*X show propagation to each method from variable occurrences inside the same class for example, we see that there are 2 propagations to method main, 0 to method values, 4 to method calcpower, 4 to method calcmean, and 4 to method output. Finally, V*Z*X*C represents the complexity weighted total variable definition propagation for each method.

The ripple effect measurement can highlight high ripple effect methods as a serious problem. These highlighted methods can be used later in regression testing. According to [KGH94] the regression process consists of 5 phases:

1. Identification of changed classes
2. Identification of affected classes
3. Generation of class test order
4. Selection of test cases
5. Test cases modification and generation.

The ripple effect can be used to identify the impact of change for each class. Relation between ripple effect and regression testing is discussed in chapter 6.

5.2 Counting Ripple effect for Calculations.java class:

The following is another example used in the computation of ripple effect and logical stability of a class called *Calculations.java*. A listing of the class source code is given in Figure 5.2

STEP 1: Make a list of all occurrences of variables (not including variable declarations) for each method in the order in which they appear in the source code. For example, in method *addnum* the list would be:

number, number, sum, sum, x, sumofsquares, sumofsquares, x.

STEP 2: Form Boolean vectors for each module. We give *addnum* as an example, with '1' in the vector denoting that the variable occurrence statisfies one or more of the five conditions specified in section 3.1, and '0' that it satisfies none. By inspection of the source in calculations.java

	n_7^d	n_7^u	sum_8^d	sum_8^u	x_8^u	sos_9^d	sos_9^u	x_9^u
V_{addnum} = (	1	1	1	1	0	1	1	0)

STEP 3: Construct matrix *Zm* for each method:

	n_7^d	n_7^u	sum_8^d	sum_8^u	x_8^u	sos_9^d	sos_9^u	x_9^u
n_7^d	1	0	0	0	0	0	0	0
n_7^u	1	1	0	0	0	0	0	0
sum_8^d	0	0	1	0	0	0	0	0
sum_8^u	0	0	1	1	1	0	0	0
x_8^u	0	0	0	0	1	1	0	0
sos_9^d	0	0	0	0	0	1	0	0
sos_9^u	0	0	0	0	0	1	1	0
x_9^u	0	0	0	0	0	1	0	1

STEP 4 : Construct matrix X_{addnum}:

	addnum	getaverage	getdeviation	getsumsquares	avgpower	main
n_7^d	0	1	1	0	0	1
n_7^u	0	0	0	0	0	0
sum_8^d	0	1	1	0	0	1
sum_8^u	0	0	0	0	0	0
x_8^u	0	0	0	0	0	1
sos_9^d	0	0	1	1	0	1
sos_9^u	0	0	0	0	0	0
x_9^u	0	0	0	0	0	1

STEP 5: Construct matrix *C* representing the McCabe complexities for the methods in this class which are as follows:

$$C = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \end{pmatrix} \begin{matrix} \text{addnum} \\ \text{getaverage} \\ \text{getdeviation} \\ \text{getsumsquares} \\ \text{avgpower} \\ \text{main} \end{matrix}$$

	C
addnum	1
getaverage	1
getdeviation	1
getsumsquares	1
avgpower	1
main	2

STEP 6: Find the Boolean product of the matrices formed in *STEP 3* and *STEP 4* $Z_{addnum}X_{addnum}$:

$$\begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&1&0&0&1&1&1 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&1&0&0&0 \\ 0&0&0&1&1&0&0&0 \\ 0&0&0&0&0&1&1&0 \\ 0&0&0&0&0&1&1&0 \\ 0&0&0&0&0&1&1&1 \end{pmatrix} \begin{pmatrix} 0&1&1&0&0&1 \\ 0&0&0&0&0&0 \\ 0&1&1&0&0&1 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&1 \\ 0&0&1&1&0&1 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&1 \end{pmatrix} = \begin{pmatrix} 0&1&1&0&0&1 \\ 0&1&1&0&0&1 \\ 0&1&1&0&0&1 \\ 0&1&1&0&0&2 \\ 0&0&1&1&0&2 \\ 0&0&1&1&0&1 \\ 0&0&1&1&0&1 \\ 0&0&1&1&0&2 \end{pmatrix}$$

STEP 7: Find the product of the matrices formed in *STEP 2* and *STEP 6*, $V_{addnum}Z_{addnum}X_{addnum}$:

$$\begin{pmatrix} 1&1&1&1&0&1&1&0 \end{pmatrix} \begin{pmatrix} 0&1&1&0&0&1 \\ 0&1&1&0&0&1 \\ 0&1&1&0&0&1 \\ 0&1&1&0&0&2 \\ 0&0&1&1&0&2 \\ 0&0&1&1&0&1 \\ 0&0&1&1&0&1 \\ 0&0&1&1&0&2 \end{pmatrix} = \begin{pmatrix} 0&4&6&2&0&7 \end{pmatrix}$$

STEP 8: Find the product of the matrices formed in *STEP 2* and *STEP 5*, $V_{addnum}Z_{addnum}X_{addnum}$:

$$\begin{pmatrix} 0 & 4 & 6 & 2 & 0 & 7 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 2 \end{pmatrix} = 26$$

STEP 9: Divide this figure by $|Vmj|$, the number of variable definitions in *class Calculations to* give the measure of ripple effect for the method. Looking at V_{addnum} we can see that there are 6 '1's in the vector, we therefore divide the product of the matrices by 6 giving:

26 / 6 = 42.33

The Logical Stability for method *values* is therefore 6 /26 = 0.2

```
1:      import java.io.*;
2:     import java.awt.*;

3:     Public class calculations {

4:     int n = 0;

5:     double sum = 0.0, sumofsquares = 0.0;

6:     public void addnum (double x) {

7:     n = n+1;
8:     sum = sum + x;
9:     sumofsquares = sumofsquares x*x;

10:    }

11:    Public double getaverage() {

12:    Double average;
13:    Average = sum/number;
14:    return (average);
15:    }

16:    Public double getstandardDeviation() {

17:    Double deviation;
18:    Deviation = ((sumofquares + sum*sum/n)/n);
19:    Return math.sqrt(deviation);
20:    }

21:    Public double getsumofsquares() {
22:    return (sumofsquares);
23:    }

24:    Public double avgpower() }

25:    double powers, averagepower = 0;
26:    averagepower= getaverage();
27:    powers = pow (averagepower,2);
28:    return (powers);
29:    }

30:    Public static void main (string args[]) {

31:    double m_average, m_StandardDeviation, m_sumofsquares,
   Averagepower
32:    for (int i = 1; i <=100; i++) addDatum(i);

33:    m_average=getaverage();
34:    m_StandardDeviation =getstandardDeviation();
35:    Numbers = n;
36:    m_sumofsquares = getsumofsquares();
37:    Averagepower = avgpower();

38:    }
```

Figure 5.2 calculations.java

Chapter 6

Future Work

Previous measure and tools developed to produce ripple effect measures for procedural software have used Yau and Collofello's algorithm which is based on set theory. It was proved difficult to write simple software using this algorithm; ripple effect tools have either taken an excessive amount of time to produce ripple effect measures or needed some user intervention to make critical decisions about the source code. In our approach, we used matrix arithmetic which simplified the process of ripple effect computation. Each matrix used within the algorithm holds a particular type of information about the software under scrutiny. This makes it easier to understand what each part of the algorithm means and how the ripple effect is being computed. Initial steps we made to automate our approach and produce ripple effect for object-oriented tool (ROOT) however, for it to be used in industry as a fully working measurement tool some further work needs to be carried out. Facilitation of ripple effect computation for other Object-Oriented programming languages besides Java should be considered in the future. Further work will also include investigation into the feasibility of measuring ripple effect at different levels of abstraction. So far we have looked at computing ripple effect using source code at class and architecture level. Ripple effect computed at subsystem and system level could provide valuable information for use by both developers and maintainers.

As a future work we suggest to develop a parser that will pick up the data from object-oriented code and used it in calculating the ripple effect automatically. The parser also needs more work to enable all java programs of any version to parse. As it has proved difficult in the past to produce ripple effect automatically for Object-Oriented Programs, no comprehensive studies have as yet been carried out using automated ripple effect tools to discover how much use ripple effect is to maintainers.

6.1 Ripple Effect of Distributed Systems

Computation of ripple effect for distributed systems could be our next target. Initial investigation was made to study if it is both meaningful and feasible to measure the ripple effect for distributed systems.

6.2 Ripple Effect and Regression Testing

The objective of the ripple effect is different from that of regression testing. Regression testing ensures that those features in system functionality that should not be changed remained unchanged after a code change. On the other hand, the ripple effect analysis is to ensure that those parts of the software that need to be changed due to dependency to the changed code are identified and changed. The ripple effect should be performed in addition to regression testing because ripple effect analysis and regression testing complete each other. Even though the objective of the ripple effect analysis and regression testing are different, it is possible to use ripple effect to minimize test cases in the regression testing where the ripple effect is used to identify those parts of the software and their corresponding test cases that are potentially affected (Impact Analysis).

The ripple effect analysis can be an integrated part of regression testing and regression test cases can be selected based on the ripple effect analysis results.
For example, Scenario based functional regression testing which is based on end-to-end integration test scenario.

1) The test scenarios are represented in a template model that embodies both test dependency and traceability.
2) Traceability information shows affected components and associated test scenarios and test cases for regression testing.
3) Ripple effect analysis can be used to identify all directly/indirectly affected scenarios and thus the set of test cases can be selected for regression testing.

Also, Ripple Effect Analysis can be used in Prioritization techniques. Prioritization techniques schedule test cases for execution in an order that attempts to maximize some objective function. (Maximize logical stability which means minimize ripple effect).

6.3 Ripple Effect Using UML

Ripple Effect Analysis can be computed using the Unified Model Language.

Using a class diagram we can determine:

1) added/deleted attribute
2) changed attribute
3) added/deleted method
4) changed method
5) added/deleted relation
6) changed in relationship
7) added/deleted class
8) changed class

Using sequence diagram we can determine:

1) added/ deleted use case
2) changed use case
3) added/deleted method
4) changed method

Change in two versions of same class diagrams:

Two version of the same diagram can be compared together to detect the sets of added, changed and deleted attributes, method relationships and classes.

Added/deleted attributes: an added/deleted attribute is an attribute that is not declared in the original/modified version of a given class but is declared in the modified/original version of this class.

Changed attributes: this exists in both version of a given class but with a different scope type or visibility.

Added/deleted method: it's a method that does not exist in the original/modified version of a class but exist in the modified/original version of this class.

Changed methods: it has the same signature in the two class diagram version but a number of other design changes translate into changed methods.

6.4 Possible Class-Level Ripple Effect Analysis estimation with syntactic impact

To calculate the ripple effect of a class (say C), the algorithm applies all possible class-level changes with syntactic impact (listed in table 1 and 2) to the attribute and methods of all other classes in the design. Changes are applied one at a time and they are all applied to the original design. For each change, the algorithm computes the set of syntactically impacted classes and determines whether or not class C is among them. The number of times that class C is found impacted divided by the total number of possible changes represents the likelihood that class C will be change-prone as a result of a class-level made else where in the design. As a result, the logical stability of class C is simply one minus that ratio. Thefore, Ripple Effect = 1/ Logical Stability. Given that types of changes are unpredictable and that no studies found that

some changes are more likely than others, it was assumed that all types of class-level changes are equally likely.

Attribute (A) of Class (C)

Change Type	Syntactically Impacted Classes
Data type	X
Delete	X
Scope (Public to private)	X – {C}
Scope (protected to private)	X – {C}
Scope (Public to protected)	X – [Z U {C}]

Table 6.6

Method (M) of Class (C)

Change Type	Syntactically Impacted Classes
Return Data type	Y
Signature	Y
Delete	Y
Scope (public to private)	Y – {C}
Scope (Protected to private)	Y – {C}
Scope (Public to protected)	Y – [Z U {C}]

Table 6.7

Where:

- X is the set of classes, including C, that reference A
- Y is the set of classes, including C, that invoke M
- Z is the set of direct and indirect subclasses of C

Algorithm:

ClassRippleEffect (Class C, ObjectOrientedDesign OOD)

Input : Class C, Object-Oriented Design OOD in which class C exists.
Output : The likelihood that C will not be change-prone as a result of a Class-level change with syntactic impact made to another class in OOD.

```
Begin

        TotalNumOfChanges = 0
        NumOfChangesImpactedClassC = 0
        FOR each class A expect C in OOD DO
          FOR each attribute T in A DO
            FOR each type of attribute change I DO (listed in table 6.6 above)
                IF I can be applied to T THEN
                   IC = {set of classes impacted by applying I to T}
                   IF C belong to IC THEN
                        NumOfChangesImpactedClassC ++
                   ENDIF
                   TotalNumOfChanges ++
                ENDIF
            ENDFOR
          ENDFOR
          FOR each method M is A DO
              FOR each type of method change J DO (listed in table 6.7 above)
                IF J can be applied to M THEN
                   IC = {set of classes impacted by applying J to M}
                   IF C belong to IC THEN
                        NumOfChangesImpactedClassC ++
                   ENDIF
                   TotalNumOfChanges ++
                ENDIF
              ENDFOR
           ENDFOR
        ENDFOR
     ENDFOR
ClassLogicalStability = (1 – (NumOfChangesImpactedClassC /
TotalNumOfChanges))
RETURN ( 1 / ClassLogicalStability)
END
```

A future work could be implementing this algorithm and comparing the results with the results of the algorithm suggested earlier in this thesis.

6.5 Ripple effect in web pages

The internet is quietly becoming the body of the business world, with web applications as the brains. Websites are something entirely new in the world of software quality. This means that software faults in web applications have potentially disastrous consequences. Within minutes of going online, a web application has many thousands more users than conventional non-web applications. Most work done on web applications has been on making them more powerful, but relatively little has been done to ensure quality. The technical complexities of a website and variances in the web browser make testing and quality control more difficult. Web applications share some characteristics of client-server, distributed and traditional programs; however, there are a number of novel aspects of web applications. These include the fact that web application are "dynamic", due to factors such as the frequent changes of the application requirement as well as dramatic changes of the web technologies, the fact that the roles of the clients and servers change dynamically, the heterogeneity of the hardware and software components, the extremely loose coupling and dynamic integration, and the ability of the user to directly affect the control of execution. Measuring the ripple effect and logical stability of a website would therefore help maintainers to achieve a more reliable websites.

Chapter 7

Conclusion

In this thesis we have invested and studied ways and methods for calculating the ripple effect for object oriented software. We have also introduced a new technique that will help in implementing an object-oriented software measurement which we hope it will be beneficial to the software maintainer and developer by computing ripple effect automatically. Measurement of ripple effect has been incorporated into several software maintenance models to give maintainers valuable information about the code they are maintaining. Maintenance is difficult because it is not clear where modifications have to be made or what the impact will be on the rest of the source code once those changes are made. The ripple effect can be used to help maintainers with assessing that impact. Along with many other metrics, ripple effect is not the answer to all maintainer's problems, but used as part of a suite of metrics it can give maintainers useful information to make their task easier. Ripple effect is not only useful during software maintenance. During software development it can be compared for different versions of a program to ascertain whether stability is increasing or decreasing and changes made accordingly.

Several ripple effect tools are already in existence for procedural software only some of which compute ripple effect without taking intramethod change propagation into account. Others are only semi-automatic, user intervention being required at some point in the computation.

Our motivation in this research was to provide a ripple effect measure for object-oriented program using matrix arithmetic quickly and completely automatically. In order to calculate the ripple effect for object-oriented programs we studied all object-oriented dependencies, relations and propagations inside and outside the class. We had also studied object-oriented complexity metrics with their relation with ripple effect and divided them into inter-class metrics and intra-class metrics

after adding new object-oriented metrics. Our algorithm calculate the ripple effect for object-oriented programs at the code level by calculating both intra-class propagation and inter-class propagation for each class, and at architecture level by calculating the ripple effect at the system level. In addition, the algorithm clarifies the process of computing ripple effect. Each matrix used within the algorithm holds a particular type of information about the software under scrutiny. This makes it easier to understand what each part of the algorithm means and how the ripple effect is being computed.

We applied our method for calculating ripple effect on 2 examples: example.java and calculations.java and explained step by step how the ripple effect is calculated using matrix arithmetic then we used the ripple effect to calculate an index for logical stability. We found that as ripple effect increases the logical stability of this program decreases which means that methods with higher will face more problem when being updated. The ripple effect measurement can also highlight high ripple effect methods as a serious problem where these highlighted methods can be used later in regression testing.

However, our suggested techniques for computing the ripple effect of local changes have limitations. One limitation is the high cost of the computational method; the barometer is the computation of the propagation matrix, which is of the order of the square of the number of variable occurrences. The second limitation is concerned with the program dependences; only 'straightforward' static dependences have been considered and semantic and run-time dependences have been ignored. Further work can address these limitations and can consider the use of the ripple effect measure in other software engineering problems.

References

[Ben90] K. H. Bennett. "An introduction to software maintenance". *Information and Software Technology*, 12(4):257-264, 1990.

[Bla01] S. Black. "Computing ripple effect for software maintenance". *Software Maintenance: Research and Practice*, 13(4), pp. 263-278,2001..

[BLS02] L.C. Briand, Y. Labiche and G. Soccar. "Automating Impact Analysis and regression Test Selection Based on UML Designs". *Carleton Technical Report SCE-02-04*, 2002.

[Boe87] B. Boehm. "Software Engineering". *IEEE Transactions on Computers*, 12:1226-1242, 1987.

[Cha84] S. C. Chang. *A unified and efficient approach for logical ripple effect analysis*. PhD thesis, Dept. EECS, Northwestern University, Evanston, Illinois, June 1984. 94 pp.

[CK91] S.R Chidamber and C.F. Kemerer, "Toward a metrics suite for object-oriented design", *Proc. Sixth OOPSLA Conference*, pp 197-211, 1991.

[CKL99] M. A Chaumun, H. Kabailim, R. K. Keller and F. Lustman. "A change Impact Model for Changeability Assessment in Object-Oriented Software Systems". *In Proceeding of the Third Euromicro Working Conference on Software Maintenance and Reengineering*, pages 130-138, Amsterdam, The Netherlands, March 1999.

[CLT96] G. Canfora, G. A. Di Lucca, and M. Tortorella. "Controlling side-effects in maintenance". *Proceedings of the 3rd International Conference on Achieving Quality in Software*, pp 89-102, 1996.

[CW87] J. S. Collofello and D. A. Wennergrund. "Ripple effect based on semantic information". *Proceedings AFIPS Joint Computer Conference*, 56:675-682, 1987.

[Han72] F. M. Haney. "Module connection analysis-a tool for scheduling of software debugging activities". *Proceedings Fall joint Computer Conference*, pp. 173-179, 1972.

[Hsi82] C. C. Hsieh. *An approach to logical ripple effect analysis for software maintenance*. PhD thesis, Dept. EECS, Northwestern University, Evanston, Illinois, June 1982. 206pp.

[IE90] IEEE. *Standard Glossary of Engineering Terminology*. Institute of Electrical and Electronic Engineers: New York NY, 1990.

[JT93] J. K. Joiner and W. T. Tsai. "Ripple effect analysis, program slicing and dependence analysis". TR 93-84, University of Minnesota technical report, 1993.

[KGH94] D. Kung, J. Gao, P.Hsia, F.Wen. Y. Toyoshima, and C. Chen, "Change Identification in Object Oriented Software Mainteance", *Proceeding of the Internatuon Conferance on Software Maintenance*, pp.201-211, 1994

[KKL02] H Kabaili, R Keller and F Lustman, " A change Impact Model Encompassing Ripple Effect and Regression Testing", *Technical Report* University of Montréal. 2002

[LI98] L. Li "Change Impact Analysis for Object-Oriented Software". PHD thesis, George Mason University, Virginia, USA, 1998.

[LO96]L. Li and A. J. Offutt. "Algorithmic Analysis of the impact of changes to Object-Oriented software". In ICSM96, pp. 171-184, 1996.

[LOA00] M. Lee, J. Offutt and R. Alexander. "Algorithmic Analysis of the impact of changes to Object-Oriented software". In *TOOLS USA '00*, pp. 61-70, August 2000.

[McC76] T. J. McCabe. "A complexity measure". *IEEE Transactions on Software Engineering*, 2(4),pp.308-320, 1976.

[Mye80] G. J. Myers. *A model of program stability*, pages 137-155. Van Nostrand Reinhold Company, 135 West 50th Street, NY 10020, 1980. Chapter 10.

[PB90] S. L. Pfleeger and S. A. Bohner. "A framework for software maintenance metrics".*IEEE Conference on Software Maintenance*, pp. 320-327, 1990.

[Pre94] R. S. Pressman. *Software Engineering: A Practitioner's Approach, (3rd edn.)* European Adaptation, McGraw-Hill International (UK) Ltd., Maidenhead, UK, 1994.

[PY73] F. Preparata and R. Yeh. *Introduction to discrete structures for computer sicnece and engineering.* (1st edn.) London: Addison-Wesley publusing company, 1971.

[RAJ01] V. Rajlich. "Propagation of changes in Object Oriented Programs". Technical Report, University of Wane State.2001

[RT01] B. G Ryder and F. Tip. "Change Impact analysis for Object-Oriented programs". In Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, pp. 46-53, Oct. 2001.

[She93] M. Shepperd. *Software Engineering Metrics Volume I: Measures and alidations*. McGraw Hill International, London, UK, 1993.

[Soo77] N. L. Soong. "A program stability measure". *Proceedings 1977 Annual ACM conference*, pp. 163-173, 1977.

[TM94] R. J. Turver and M. Munro. "An early impact analysis technique for software maintenance". *Software Maintenance: Research and Practice*, 6: pp.35-52, 1994.

[VZE93] V. Zuylen (ed.) *The REDO compendium*. Wiley: Chichester, UK.1993.

[YC80] S. S. Yau and J. S. Collofello. "Some stability measures for software maintenance", *IEEE Transactions on Software Engineering*, SE-6(6):545-552, 1980.

[YC84] S. S. Yau and S. C. Chang. "Estimating logical stability in software maintenance". *Proceedings COMPSAC '84*, pp 109-119, 1984.

[YCM78] S. S. Yau, J. S. Collofello, and T. M. McGregor. "Ripple effect analysis of software maintenance". *Proceedings COMPSAC '78*, pp 60-65, 1978.

[YL88] S. S. Yau and S. Liu. `Some approaches to logical ripple effect analysis'. SERC- TR-24-F, University of Florida technical report, 1988.

[Wei84] M. Weiser. "Program slicing". *IEEE Transactions on Software Engineering*, SE-10(4):pp.1352-1357, 1984.

[W*et al*96] Y. Wang and W. T. Tsai *et al*. "The role of program slicing in ripple effect analysis". TR 96-014, University of Minnesota technical report, 1996.

[WH92] N. Wilde and R. Huitt, "Maintenance Support for Object-Oriented Programs," *IEEE Transaction Software Engineering*, 18(12), pp.1038-1044.December 1992.

[WK00] F.G. Wilkie and B.A Kitchenham. "Coupling measures and change ripples in C++ application software". The Journal of Systems and Software, 52: pp.157-164, 2000.