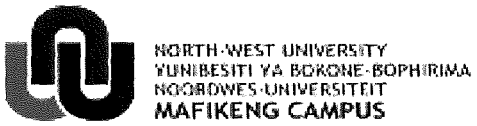
Early Failure Prediction During Change Impact Analysis For Improving Object-Oriented Software Maintenance

North-West University
Mafikeng Campus Library

BASSEY ECHENG ISONG

(Student No: 24073008)
BSc (Hons) CS., MSc. CS. & MSc. SE.

A Thesis Submitted in Fulfillment of the Requirements for the award of the Degree of Doctor of Philosophy (PhD) in Computer Science



LIBRARY
MAFIKENG CAMPUS
Call No.:
2014 -10- 20
Acc. No.: 14/6345
NORTH-WEST UNIVERSITY

Department of Computer Science
Faculty of Agriculture, Science and Technology
North-West University
Mafikeng Campus
South Africa

Supervisor: **Professor O.O. Ekabua**

June, 2014

DECLARATION

I declare that this research study on **Early Failure Prediction during Change Impact Analysis for Improving Object-Oriented Software Maintenance** is my work, and has never been presented for the award of any degree in any University. All the information used has been duly acknowledged both in text and in the references.

Signature:_____ Date: 18 - 09 - 14

Bassey Echeng Isong

Approval

Signature:_____ Date:_____

Supervisor: **Professor Obeten Obi Ekabua**
Department of Computer Science
North-West University
Mafikeng Campus
South Africa

DEDICATION

Mom, Dad and Son - Kelvin

ACKNOWLEDGEMENTS

Life is awesome when God is in control and good relationships shape everything we do. The research work reported in this thesis would not have been possible without the support and influence of a number of wonderful people whom I had the opportunity to meet.

First of all, I am grateful to Prof. Obeten Obi Ekabua, my supervisor, for his invaluable support throughout the research. His advice, guidance, ideas and supports have been instrumental in piloting this research. This thesis would not have been a reality without his constant support and encouragement. I therefore, say a big thank you for allowing God to use you in achieving this dream in my academic pursuit.

I want to thank the University of Venda, Thohoyandou and the North-West University, Mafikeng, both in South Africa, for affording me the opportunity and financial assistance to undertake this Doctoral degree. I am also thankful to all the staff members of the Department of Computer Science in Univen and NWU, and the members of my research team, especially Ifeoma Ugochi Ohaeri and Nosipho Dladlu for their valuable support.

I remain absolutely indebted to my family for providing me with the invaluable support, confidence and comfort to undertake this research across the border. I would like to thank Edu, Kelvin, Mom, Dad, Innocent, Kingsley, Bridget, Anthony, Peace and others. I would not have desired a better family than you are to me.

Lastly, I would like to thank God Almighty who made it possible for me to complete this work against all odds. There is nothing He cannot do and His Glory cannot be shared. Thank You Father!

TABLE OF CONTENTS

NWU LIBRARY

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Change is inevitable and an important property of software. Software applications are changed during their life-time in order to remain useful. Nonetheless, changes also come with high risks when it is made. Regardless of their size, they can have significant and unexpected effects elsewhere in the software, degrade software quality or cause them to fail. Change impact analysis is used to preserve software quality. Today, as object-oriented technology has gained worldwide popularity, several object-oriented software applications are currently in use. Given the critical context, it is important that these systems are effectively and efficiently maintained if continuous usefulness is the goal. However, object-oriented paradigm introduces specific features, have different change and complex dependencies types often which makes it hard to identify the impact of changes or it is likely that they might introduce some types of faults which are difficult to detect. In addition, the available impact analysis techniques offer litter or no information on explicit program representation, they are not precise and produce large impact sets which are not good for practical use. Hence, an effective technique that can precisely predict true impact set and identify early enough, which components affected by a change are fault-prone is needed. This is necessary to reduce the risk associated with field failures when changes are made. Moreover, traditional research on software change impact analysis and fault prediction is disjointed. Therefore, in this research work, we design a change impact analysis framework that incorporates both impact and early fault prediction in the maintenance of object-oriented software. The objective is to enhance program comprehension, reduce the time, effort and the risks associated with software change while software quality is preserved. We achieved this by exploring and analyzing object-oriented programs complex relationships, using intermediate source code representation that explicitly reveals their implicit structure, dependencies and allow for complexity quantification in small to medium sized systems. The representation alongside the impact diffusion range of a given change type is used to predict change impact and improve its precision. Additionally, logistic regression was used to build an early fault prediction model which utilizes object-oriented product and process metrics. The approaches were empirically evaluated and the results obtained showed that the source code representation is effective and practical for impact analysis and the change impact analysis technique showed improved precision. Also, the fault prediction model shows high accuracy, sensitivity and specificity. To facilitate the prediction process, this research implemented a novel tool called ClassChangeRecommender to assist software maintainers in predicting which components impacted by a change are fault-prone to allow mitigation action in advance before actual changes are made.

LIST OF ABBREVIATIONS

Acronym	Meaning
CIA	Change Impact Analysis
SIS	Starting Impact Set
EIS	Estimated Impact Set
TIS	Total Impact Set
AIS	Actual Impact Set
OO	Object-Oriented
SDLC	Software Development Life Cycle
LR	Logistic Regression
IR	Intermediate Representation
SCM	Software Configuration Management
CVS	Concurrent Versioning System
RO	Research Objective
SO	Software Object
CP	Change Proposal
PC	Program Correctness
NoE	Number of Errors
SLR	Systematic Literature Review
CLR	Comprehensive Literature Review
OOComDN	Object-oriented Component Dependency Networks

CHAPTER 1

Introduction and Background

1.1 Introduction

Software development is a complex and difficult activity that involves the transformation of stated customers, users or market needs into a final product. These needs are the requirements which go through a series of development activities such as architecture, design, implementation and testing to form the final product that is delivered to the owner. However, software development does not stop when a system is delivered, but has to continue throughout the lifetime of the system, especially large software systems. After a system has been deployed, change becomes inevitable if it is to remain useful [1]. This is because software systems are critical assets or resources for organizations. Organizations invest huge amounts of resources in their software and they are completely dependent on it. Therefore, investing in system change is necessary to maintain the value of these assets, as the longer the software application supports the needs of the organization, the more successful it is. This is evident in most large organizations today as they spend more on maintaining existing systems than on developing new systems.

Software changes being inevitable in software development, is a key operation for evolution. Unlike several other types of products, a software product is intended to be malleable and adaptable in order to continue fulfilling its user and operational requirements [1][2]. Drivers of software changes are the addition of new requirements, error corrections, change requests, restructuring of the software to accommodate future changes, performance improvement, and so on [3][4]. In particular, changing requirements are common, and this is one of the most significant motivations for software change. Nonetheless, regardless of size of the change, it can have considerable and unexpected effects on the system. In some cases, it may lead to software deterioration or introduce faults in the software if not well-understood. The fact remains that, albeit software does not deteriorate or change with age, most software maintenance involves change that potentially degrades the quality of the software unless it is proactively controlled [2][5]. Thus, system modification needs to be taken seriously and changes impacts must be considered as well. Change impacts are indirect and difficult to discover, consequently mechanisms are required to analyze changes and to know how they are propagated in the entire system.

CIA is a technique that is used to understand and identify the potential effects caused by changes made to software [2][6]. Given a reasonable understanding of the software, the objective of CIA is to understand how a proposed change in the implementation will affect the software components. Like requirements, changes can also be made to source code, in particular, objected-oriented (OO) software code. In this case, the affected components are not only the classes, methods or functions and fields, but also the design models, user manuals, test cases and so on. An effective CIA can improve the accuracy of required resource estimates, allow more accurate development schedules to be set, and reduce the amount of corrective maintenance by reducing the number of errors introduced as a by-product of the maintenance effort [5][7]. In a nutshell, these improvements can result in the reduction of risks, efforts and costs associated with the proposed changes [5]. Without CIA and management mechanisms, software changes during maintenance can have unpredictable consequences that will delay their implementation. Thus, this thesis deals with CIA from the perspective of OO program source code.

1.2 Background Information

One important property of any software is change [5]. Changes occur in every phase of software development such as requirements, design, implementation, testing and maintenance. Despite their importance, changes also come with potential risks. Firstly, changes in one phase of development can affect the behaviour of the delivered software product in another phase. Secondly, when changes are made to software systems (e.g. source code), regardless of the change size, they have the ability to introduce unanticipated potential effects and errors elsewhere in the software system. It may also introduce inconsistencies to other parts of the software due to derived changes which are directly or indirectly affected by the change, degrade the quality of software or cause the software to fail. In such a situation, the changed and the affected components may no longer be compatible with the rest of the software product, a situation that leads to software deterioration [5]. Software deterioration occurs in many cases because changes to software rarely have the small impact they are believed to have [5]. This stems from impact overlooking, impact underestimation and impact overestimation.

Today, software applications have grown in size and complexity, incorporating more features and newer technologies. Their dependency webs are believed to have extended beyond most software engineers' ability to comprehend [2]. Consequently, due to a huge amount of information coupled with inadequate comprehension of the program, many ripple-effects of software change can go unnoticed until they are noticeable in system failure. In particular, changing source code in large software applications today is very difficult and requires a good understanding of the

dependencies between the software components. This is because making changes to software components with little or no regard to their dependencies may have unexpected effects on the quality of the latter which may increase their risk of failure. For software change to be effective, program comprehension is indispensable, in order to identify indirect impacts which result from the affected components. This is important to maintain the consistency and integrity of the software product after changes have been made. In this case, understanding the nature of the changes to be implemented allows more effective prioritization of change requests [3]. Research and experience has also shown that making changes without understanding their effects can lead to poor estimates of effort and decision making, delay in release schedules, degraded software design, unreliable software products, the premature retirement of the system and consequently failure [7].

CIA is a process for controlling changes and avoiding software deterioration if properly applied. It plays a crucial role in various software maintenance activities, which can be used before or after change implementation. Before changes are made, CIA can be utilized for program understanding, change impact prediction, cost estimation and so on [6]. After changes have been implemented in the original system, CIA can also be applied to guide regression, select test cases, and perform change propagation and ripple effect analysis [5][6]. The cost of the change can be used to decide whether or not to implement it depending on its cost/benefit ratio [5]. For instance, if a change is known to impact every part of the system, the decision would be to avoid this change, as the cost would be huge.

As CIA plays a vital role in the maintenance of software applications, OO programs are not exceptions. In the last decade, OO approaches have become a dominant approach in software design and development and several OO applications are currently in use today. Therefore, the systems have to be effectively maintained. The popularity of OO software stems from the benefits of better maintainable, reusable systems and efficient component-based development [7][8]. These are as a result of many specific useful features which often differentiate it from the function-oriented paradigm such as *encapsulation*, *inheritance*, *information hiding*, *polymorphism* and *dynamic binding* [8]. Unfortunately, these features frequently lead to more complex relationships among classes and understanding the relationship in order to make changes is very challenging. The complex relationships make it difficult to know in advance or identify the ripple-effects of changes [7][8]. An object has state and behavior and different dependencies: inheritance, membership, invocation and usage. These make it hard to define a cost-effective test and maintenance approach [7]. Therefore, a good analysis method is required for OO programs. With an effective CIA method, one can determine for some level of granularity which

components in the software are truly affected by changes. This will result in a reduction of corresponding efforts, risks and costs that accompany the change request.

In addition, OO software features can also cause some types of faults that are difficult to detect using traditional testing approaches. Empirical evidence has shown that most OO programs are fault-prone or failure-prone [9][10][11]. Faults in software applications are believed to be found in only a few of a system's components. If these faults are not detected on the affected components before changes are made, it could result in software failure. Identifying these components will allow mitigating actions such as validation and verification activities to be focused on the high risk components so as to avoid the risk associated with field failures [9][11]. Hence, predicting this risk early can be effective in improving software maintenance while preserving its quality. This thesis, therefore intends to evolve a failure prediction model that will be incorporated into the CIA techniques for effective decision making during software changes.

1.3 Problems Statements

In this section, we state the problem statements and how they will be addressed in this research work.

1. Software CIA constitutes one of the most tedious and difficult tasks of software change. Given a proposed change, the task of CIA is to determine the potential effects of the change on a subject system, in this case, OO source code. The input is the *change set* while the output is the set of components thought to be affected by the change called *impact set* [2]. There are several CIA that exist today in the perspective of OO programs which are either based on *static* CIA, *dynamic* CIA or a *hybrid* CIA [12]. Each of these methods has its own strengths and weaknesses. Taking static CIA methods into consideration, though they utilizes call and program dependencies graphs for the analysis, they are known to be safe but *less precise*, and *produce very large impact sets which are difficult for practical use* [2][12]. Existing static CIA methods have not sufficiently addressed this challenge.
2. The complex relationships resulting from features specific to OO programs often make it difficult to anticipate and effectively identify the ripple-effects of changes. In addition, there are different change and dependencies types, with each having its own impact diffusion range. For instance, some changes made to a program component do not affect other components in the programs in spite of some dependencies that exist between them, while some changes may potentially impact other entities in the program [12]. This poses a huge challenge to maintaining OO programs.

3. Also, OO program components despite the benefits OO program are not immune to being faulty or failure-prone [9][10][11]. A software fault can cause an executable product to fail either during testing or in the field. In large software systems, the early identification of these components will allow efforts to be channelled towards the high risk components in order to avoid the risk associated with field failures when changes are actually implemented. If these classes are not detected before changes are made, it will increase the likelihood of software failure. In this case, regression testing and other development activities will be negatively affected. Therefore, the choice of a suitable model for such an analysis forms part of this research and predicting faults early during CIA is important.

In this thesis, based on the characteristics of static CIA and OO programs, static CIA is considered the most suitable CIA method for analysing the impact of a change in OO source code. The important questions are: 1) how to improve the precision and reduce the computed impact set of static CIA methods, 2) how to construct an approach that explicitly reveals the implicit structure of OO program components as well as 3) predict which classes affected by a change will be faulty if changes are implemented on them. In order to contain the inherent complex relationships among OO components and to understand them for successful change implementation, an effective static CIA is of the essence. An effective static CIA method will be able to effectively predict the impact of a change, and predict early the risks associated with the change implementation.

Several works on CIA and software prediction have been reported but they focus on predicting classes that will change during impact analysis and maintainability [14]. No known work exists from the point of view of CIA and early failure of classes during impact analysis. Therefore, the main motivation of this work is to improve the maintenance of OO programs, and it involves more specifically CIA and potential class failures when changes are made. This research will address the challenges of CIA for OO programs and provides an approach that will reduce or eliminate the risks associated with change implementation to avoid costly software failure. In this case, the problems stated in 1 and 2 will be addressed by using IR of OO program through complex software networks, change and dependencies types and their *impact diffusion*. The problem stated in 3 will be addressed by computing the faults propagation of OO program components using complex software networks for small to medium sized systems while fault prediction model will be constructed using logistic regression and software metrics for large software systems.

1.4 Research Questions

In presenting the research questions to be answered in this thesis, the guidelines given by Creswell [13] were followed. According to Creswell [13], when stating research questions, it is worthwhile to state one or two main questions and four to seven sub-questions. In this case, the main research questions (MRQs) for this work are presented and described and sub-questions are then listed below them:

> **MRQ1:** *With the complexity associated with the relationships existing in object-oriented programs, how can we perform change impact analysis that will effectively capture the relationships and reduce the impact sets for successful change implementation?*

This question was formulated in order to gain insight in to how OO programs can be represented to explicitly reveal their implicit structures and dependencies as well as to compute their complexity quantitatively. In addition, we want to discover how changes made to a component affect other components connected to it directly or indirectly as well as to devise an approach that will help to improve static CIA method precisions while reducing the computed impact set. This will involve the use of effective intermediate code representation (IR), change and fault propagation model and framework. For effectiveness and completeness in answering MRQ1, the following sub-questions are formulated as follows:

> ***RQ1.1*** *How can we represent OO programs effectively?*
>
> ***RQ1.2*** *How does a particular change and dependency type in an OO program affect other components they are connected to directly or indirectly?*
>
> ***RQ1.3*** *How can static CIA method be used to improve the precision of the predicted impact set?*

RQ1.1 is answered in Chapter Four, while RQ1.2 and RQ1.3 are answered in Chapter Five.

> **MRQ2:** *Can we develop a change propagation framework that will predict early, the failure or the risks associated with change implementation based on the predicted impact sets?*

This question stems from the need to support CIA in large-scale software applications using software metrics. It involves the identification of OO complexity metrics and process metrics which have been empirically validated as being related to fault-proneness of classes. The metrics will then be used to construct a fault or failure prediction model that will be used to predict the fault-proneness of classes affected by a change request. To answer this question, the following sub-questions are formulated:

RQ2.1. *Which metrics are suitable for OO program's fault prediction?*

RQ2.2 *Based on the metrics, how can we formulate a model that predicts the early failure of components which are affected by change request?*

In this case, RQ2.1 is answered in Chapter Six while RQ2.2 is answered in Chapter Seven.

1.5 Research Rationale

In the realm of software development today, OO technology is becoming a *de facto* standard of development and several OO software systems are currently in use coupled with the growing popularity of OO programming languages, OO tools, OO metrics and so on. Therefore, it is imperative that these systems are maintained effectively and efficiently. As an OO program is believed to have complex relationships that often affect its maintenance, some classes that are fault-prone which could cause software failures, and the CIA techniques employed are not precise, it is essential to have an effective CIA method in place that can predict change and faulty components prior to change. An approach that is effective at analysing and capturing the complex dependencies between OO software components, as well as predicting early enough the risk associated in implementing changes in a software component, is indispensable. With this approach, the maintenance efforts, risks and costs can be reduced while ensuring the quality of the software. Effort can also be reduced if software behaviour can be predicted in the face of possible changes, and well-informed decisions can be taken before changes are made. In addition, by identifying the potential impact of the changes, the risk of dealing with costly and unpredictable changes will be reduced.

The fact remains that several CIA approaches for OO software and fault prediction models exist but these two approaches are conducted separately and no link exists between them. Because there is no known existing CIA method that incorporates change impact and fault prediction together, this thesis aims to fill this research gap. In addition, the CIA method discussed in this thesis will be used to teach undergraduate students how to perform OO program maintenance. This is because, currently, the teaching and learning of Software Engineering Education has been centred on coding and not much has been done on software maintenance. As students are the future of software organizations, it is important that they are equipped with the core competencies and skills required to maintain OO software professionally when they start working in the software industry.

1.6 Research Goal and Objectives

1.6.1 Research Goal

The main goal of this thesis is to design a CIA framework and model for early failure prediction of the impact of changes to OO programs to enhance software quality and reduce the cost, effort and risks associated with its maintenance.

1.6.2 Research Objectives

To achieve the goal of this thesis, the following research objectives (ROs) are essential:

- RO1: Analyzing the role of software change impact analysis in software maintenance by constructing an IR which explicitly reveals the structure and dependencies of the OO program.
- RO2: Determining the ripple effects of change impact of OO program according to their change type and component dependencies to obtain impact set of changes as results.
- RO3: Review current OO metrics and process metrics to determine which ones are suitable for predicting fault–prone components or classes which can trigger software failure if changes are made in such classes.
- RO4: Formulating a fault prediction model based on the metrics that will predict which class in the predicted *impact sets* will be fault-prone. In the light of the volume of large software applications today, where testing is known to be time-consuming, predicting which classes will be fault-prone will help mitigating actions to be taken on such classes and will, in turn, help reduce the risk associated with OO program change implementation.
- RO5: To design a change propagation framework that incorporates change impact and fault prediction to enhance successful change implementation in an OO program.

1.7 Research Methodology

When conducting research, the use of appropriate research methodology is very important. Correct methodology is necessary to provide clarity and transparency in terms of research reporting methods and procedures in order to responsibly show how data have been collected, synthesized, analyzed and discussed [13]. In addition, it should be possible to replicate studies if necessary, and to assure the trustworthiness of the results.

In this section, we describe the various research methodologies used in this thesis, the thesis approaches, and how trustworthiness is achieved in the various thesis chapters. The thesis employed a mixed research methods approach, *quantitative* and *qualitative* approaches [13]. Accordingly, since this thesis assumed a framework design for change impact and failure prediction as the main approach, the following approaches were employed to justify the research methods chosen: Literature survey, formulative and empirical research approaches which are all in line with Zelkowitz and Wallace's taxonomy for software engineering validation [15]. There are explained in the following subsections.

1.7.1 Literature Survey Approach

This approach involved review and analysis of related literature. In this regard, we used both comprehensive literature review (CLR) and systematic literature review (SLR). Although some published literature about CIA of software requirements and source codes exists, it mainly concerns change impact prediction coupled with less precision and large impact sets. This research argues that, if OO program components are faulty or failure-prone, there is the implicative tendency that the predicted impact sets will be fault or failure-prone. A change not intended to correct the existing faults on such classes may result in software failure and can delay other development activities such as release planning, change implementation and delivery. Analysis of related literature will help to build a substantial CIA approach that is more precise and predictive in nature towards applying this CIA technique onto OO programs. In addition, CLR will also be used to collect process metrics while SLR will be used to collect OO metrics which have been empirically validated as having influence on class fault-proneness to be used as predictors in the construction of the prediction model.

1.7.2 Formulative Approach

Based on the knowledge and information obtained from the literature reviews and component analysis, the elements of the CIA technique can be formulated using 1) CIA framework design, 2) model formulation.

1.7.2.1 Framework Design

With framework design, the existing CIA frameworks will be used as a guide to design a standard CIA framework. The available CIA frameworks will be reconfigured to incorporate risk prediction by utilizing the knowledge gained from literature of OO program features, OO program CIA and software fault prediction to design an effective OO program CIA that is precise, predictive in nature and easy to apply at all levels.

1.7.2.2 Model Formulation

In order to formulate the models used in thesis, the knowledge gained from both CLR and SLR of OO program features, CIA, process metrics (change history, fault data, and so on) and OO design metrics are used to build two models – change impact and class fault prediction. The change impact prediction model is used only to predict which OO program component will be affected if a change is made. The impact sets generated here will be based on the component *change type* and *dependency analysis* of the OO source code. On the other hand, the fault or failure prediction model will be used to predict which classes from the predicted impact sets will be fault or failure-prone. The second model depends on the output of the first model as only the impact set candidates will be used for prediction purposes.

1.7.3 Empirical Approach

Empirical experimentations in software engineering play an important role in the evolution and evaluation of a software system, tools, techniques and methods [16]. They provide a means of contributing to the body of knowledge in Software Engineering through the support of observation and empirical evidence. In this case, they allow theories to be tested, important variables to be identified and models that can be supported by empirical evidence to be built.

Based on the source code representation technique and the two models formulated in this thesis, an empirical research was also applied to evaluate the work in terms of observation and experience. The empirical strategies adopted are in line with Wohlin et al. [16] strategies for conducting empirical research in software engineering. The strategies are the experiments (*quasi* and replicated), and a case study, where the first approach supports *quantitative* methods, while the second supports both *qualitative* and *quantitative* methods. We used the *quasi*-experiment to evaluate OO program IR discussed in chapter 4 via control experiment to identify its impact on program correctness, change duration and the number of faults introduced during software modifications. A method for statistical inference was applied to show the statistical significance that IR can contribute to effective CIA. Furthermore, the case study was used to evaluate the change impact prediction model in terms of *precisions* and *recalls* discussed in chapter 5. Lastly, for the early class failure model, we employed replicated experiments or studies using NASA datasets from previous empirical studies as reported in Chapter 7 of this thesis.

1.8 Contributions

This thesis presents a methodology for conducting CIA of OO programs in terms of impact prediction and class faults or failure prediction. However, the thesis contribution is not only restricted to the methodology, but also to the findings obtained which can be used to improve the teaching and learning of software maintenance at the undergraduate level. The most prominent contributions of the thesis to software engineering discipline in general and software maintenance in particular are as follows.

a) Firstly, it offers an approach to improve the comprehension of an OO program via intermediate source code representation that explicitly reveals its implicit structure and dependencies. This work has already been published [19].
b) Secondly, it developed an approach that will allow software maintainers to perform static CIA on OO programs which will precisely predict the impact of a change and the fault-proneness of classes before changes are made. It designed a CIA framework that incorporates impact prediction and failure prediction and is also already published [17].
c) Thirdly, this thesis evolves software metrics considered to be empirically validated and good predictors of class fault-proneness or failure-proneness that can accurately predict fault or failure in an OO class early during CIA. This has also already been published [18].
d) Fourthly, we found that the CIA approach discussed in this thesis can be used to improve and foster the teaching and learning of software maintenance at the undergraduate level. This is important because most of the undergraduate students are going to be maintaining OO programs in the industry when they graduate. Thus, it is imperative they have such skills in terms of program comprehension and maintenance.
e) Finally, this thesis developed and implemented a novel tool called Class Change Recommender that will assist software maintainers in the prediction of class fault-proneness during CIA.

1.9 Included and Related Papers

This thesis builds on some research studies which have previously been reported in conference proceedings and journals. In this section, we describe a few of the manuscripts that have been incorporated in the thesis. In the papers outlined below, (Isong and Ekabua are the main authors).

I. Part of Chapter 3 is based on a paper entitled **"Towards Improving Object-Oriented Software Maintenance during Change Impact Analysis"**, published in the *Proceedings of International Conference on Software Engineering Research and Practice* (SERP'13) [17] WorldComp'13 Las Vegas, Nevada, July, 2013. The paper proposes and describes the

framework for conducting CIA on OO programs that incorporates change and failure prediction while enhancing software quality and reducing maintenance time, cost and effort

II. Part of Chapter 6 and 7 is based on a paper entitled **"A Systematic Review of the Empirical Validation of Object-Oriented Metrics towards Fault-Proneness Prediction"**, published in the *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* December, 2013 [18]. The paper conducted a systematic literature review of empirically validated CK metric suits and software lines of code (SLOC) metrics as well as on the state-of-the-art in OO fault prediction with respect to good predictors of fault-proneness, statistical techniques, tools, validation approaches and environments, programming languages, and so on.

III. Part of Chapter 4 is based on a paper entitled "**Effective Representation of Object-oriented Program: The Key to Change Impact Analysis**", published in the *Proceedings of International Conference on Software Engineering Research and Practice (SERP'14)* [19] WorldComp'14 Las Vegas, Nevada, July, 2014. The paper describes an intermediate representation of OO program for program comprehension and CIA. Its effectiveness was evaluated using student projects with respect to duration, correction and number of errors introduced during the modification task.

1.10 Thesis Structure

The structure of this thesis is described in this section.

Chapter 2 provides a comprehensive literature review of change impact analysis, object-oriented program features and software metrics and their overview in accordance with the objectives of this thesis. In addition, the chapter explains the key terminologies used in this thesis and some related studies on object-oriented impact analysis.

Chapter 3 describes concepts specific to object-oriented change impact analysis, framework and program comprehension. The chapter gives background information and outlines object-oriented concepts, change and dependencies types, program comprehension and the proposed framework for CIA.

Chapter 4 provides an IR of object-oriented software components. The chapter uses the idea of complex software networks to model OO programs (packages, classes, methods and fields) and their complex dependencies. Moreover, the IR provides a way of quantifying the complexity of the program via fault propagation and change propagation via its transformation to adjacency matrixes. The evaluation of the representation is also reported.

Chapter 5 provides an approach for object-oriented program impact prediction method and process. The chapter presents an impact model or propagation technique that is based on the change types, the dependencies types we called impact diffusion range. In addition, the chapter explains how the starting impact sets (SIS), estimated impact sets (EIS), total impact sets (TIS) and the actual impact sets (AIS) are obtained. The evaluation metrics are discussed and the effectiveness of the technique is also reported.

Chapter 6 presents the different empirically validated software metrics (product and process) which we used as predictors in the prediction model. The chapter explains how the CLR and the SLR were conducted and their results. Furthermore, the chapter provides information on how process metrics (change and fault data) can be stored and extracted in a project.

Chapter 7 describes the construction of the prediction model used in this thesis. The chapter outlines the prediction model based on the statistical technique used (Binary logistic regress), the dependent (fault fault-proneness) and independent variables (product and process metrics), variable selection techniques and evaluation in terms of accuracy, sensitivity and specificity. The chapter also presents an experiment that evaluates the fault prediction model and a novel tool for predicting the fault-proneness of classes considered to be impact set candidates.

Chapter 8 is the concluding chapter of this thesis. The chapter starts with a summary, followed by conclusions and recommendations and finally, the research limitations and suggestions for future work.

CHAPTER 2

Software Maintenance and Related Studies

2.1 Introduction

This chapter presents a general overview of impact analysis and software maintenance in the perspective of related works on *Change Impact Analysis*. It then advances to defining the basic concepts and terminologies used throughout this research. In addition, the influence of OO program features on maintenance and software measurement concepts and OO metrics are discussed.

2.2 Impact Analysis Overview

Software maintenance plays a critical role in ensuring the usefulness of software products, though it is the most difficult and costly of all the phases of software development. Software maintenance does not exist in isolation instead it incorporates the service of change, an essential ingredient to ensure that software products in which organizations have invested much do not become obsolete. Thus, change is an integral part of maintenance. However, software changes, like societal changes, do not come easily. If changes made to software systems are not properly controlled, they can result in software deterioration [5]. One instance of this is the case of Firefox Moxilla application that has up to 2 000 000 SLOC (source lines of codes) which has undergone changes that were not properly managed. Analysis shows that the software deteriorated greatly and became very difficult to maintain [5]. Several cases to support this have been reported from different industrial projects.

To avoid software deterioration, irrespective of the size of the change, the change process has to be properly controlled because it could have an unpredictable impact elsewhere in the product. This is important in today's software development because since software applications have grown in size and complexity, change has to be controlled, otherwise it could lead to deterioration or increase the risk of field failure. For instance, recent decades have witnessed the proliferation and successes of OO technology that has given birth to several programming languages such as C++, C#, Java, PHP, and Python coupled with modeling approaches and tools. OO software systems have specific features that usually result in complex dependencies among components [24]. This requires that in order to maintain them effectively and efficiently, we have to properly

quantify the impact, effort and risks of such changes to decide whether to accept or reject the change. This is where impact analysis is involved.

The word impact is a word in the dictionary that means the effect of something on another thing. In the perspective of software change, impact can be understood as change consequences. The concept of impact analysis is not new and has generally been used to assess the scope of proposed change in order to accurately estimate needed resource, plan schedules, and carry out cost-benefit analysis on the change [6]. In the Software Engineering field, the term software change impact analysis (CIA) is mainly used to evaluate which components will truly be affected if a proposed software change is implemented [5][6]. It determines the impact range and the complexity associated with the change. Thus, as stated by Lee [24], the effects that are quantitative and qualitative in nature that one change has on other components related to them directly or indirectly is what impact analysis is concerned about. For decades now, considerable amounts of research have been dedicated to impact analysis. However, there has been no agreed definition of CIA, rather it is defined based on the context in which it is used. During the 80s, work on impact analysis was specifically centered on ripple effect [20][21]. In this era, Horowitz et al [2] defined impact analysis as "*the examination of an impact to determine its parts or elements*". In another definition by Pfleeger et al [21], CIA was defined as "*the evaluation of the many risks associated with the change, including estimates of the effects on resources, effort, and schedule*". Turver and Munro [22], defined CIA as "*the assessment of a change, to the source code of a module, on the other modules of the system. It determines the scope of a change and provides a measure of its complexity*". IEEE Standard for Software Maintenance [23] defined impact analysis as a necessary activity of the software maintenance process. During 1996, Arnold and Bohner [6] defined CIA as "*the process of identifying the potential consequences of a change, or estimate what needs to be modified to accomplish a change*". Thus [6] has become the most often used and widely recognized definition of CIA.

In the perspective of software artifacts, impact analysis is an important part of requirements engineering and source codes changes. This is because changes to software often are initiated by changes to the requirements which in turn change the source code that realized the change in requirements. OO programs, the dominant in software have to be maintained in a controlled manner using CIA in order to ensure that the quality of the software is preserved during the change. The ripple-effect of a change to the source code of a software system is the rationale behind impact analysis in the context of this research.

2.2.1 Key Terminologies

In the context of this research, the following terminologies or concepts are used:

i. *Software Object (SO):* An SO is an artifact produced during a project, such as requirements, architectural components, a class and so on and is central to impact analysis.

ii. *Dependencies Analysis*: A detailed relationships among program entities, for example variables or functions, are extracted from source code. SOs are connected to each other through a web of relationships. Relationships can be both between SOs of the same type, and between SOs of different types. Dependencies tend to exert at least some influence on the overall success and quality of the product.

iii. *Change Propagation*: Inconsistencies resulting from a change perform in some part of a system. For example, changes from one object affecting other objects via dependencies and traceability links.

iv. *Side Effects*: Unintended behaviors resulting from the modifications needed to implement the change. Side effects affect both the stability and function of the system and must be avoided.

v. *Ripple Effects:* Effects on some part of the system caused by making changes to other parts. Ripple effects cannot be avoided, since they are the consequence of the system's structure and implementation. They must, however, be identified and accounted for when the change is implemented [3].

vi. *Impact*: is a measure of the ripple effect of changes that modify a software component. Impact is the average number of source code components that co-change with the source code component over the number of changes to the application in a given period.

vii. *Direct impacts*: Component set identified by analyzing how the impact of a proposed change affects the system.

viii. *Indirect impact:* It can be described in terms of ripple-effects and side effect [2][5].

ix. *Starting Impact Set:* SIS is the initial set of objects thought to be affected by a change. This is normally determined while exploring the change specification.

x. *Estimated Impact Set:* EIS is the set of objects estimated to be affected by a change. The EIS is produced while conducting the impact analysis

xi. *Actual Impact Set:* AIS is the set of objects actually modified. The AIS is not necessarily unique, as a change can be implemented in several ways.

xii. *Software Maintenance*: The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [3]. It is an expensive process where an existing program is modified for a

variety of reasons, including correcting errors, adapting to different platforms or processing environments, enhancing to add functionality, and altering to improve efficiency.

xiii. *Evolution:* Evolution is a process of continuous change from a lower, simpler, or worse to a higher, more complex, or better state [3].

xiv. *Maintainability:* The ease with which maintenance can be carried out.

xv. *Software Fault:* A defect that causes software failure in an executable SLO. It is a hidden programming error that may or may not manifest as a failure.

xvi. *Software fault-proneness:* It is defined as the probability of the presence of faults in the software.

xvii. *Software Failure:* It is a situation where the software does not do what the user expects. Every failure can be traced back to some faults, but a fault need not always result in a failure.

xviii. *Failure-Proneness:* It is the probability that a component will fail in operation in the field after a change has been made. The higher the failure proneness, the higher is the probability of experiencing a post-release failure.

xix. *Failure Prediction:* Failure prediction is to identify failure-prone situations. That is, situations that will probably evolve into a failure.

xx. *Metric:* A metric is the relationship between an attribute and its scale. It is also called a measure .Good metrics should facilitate the development of models that are capable of predicting process or product parameters.

xxi. *Object-Oriented Program*: Object-Oriented Program is a program that uses the concept of "objects" to design applications and computer programs.

xxii. *Field:* A field is a variable or parameter that is encapsulated into an object.

xxiii. *Message:* Message is a request made from one object to another to perform an operation.

xxiv. *Method:* A method is an operation upon an object, defined as part of the declaration of a class.

xxv. *Class:* A class defines the characteristics of its objects and the methods that can be applied to its objects.

xxvi. *Polymorphism:* It is the ability of two or more objects to interpret a message differently at execution, depending upon the superclass of the calling object.

xxvii. *Inheritance:* It is a relationship among classes where one class shares the structure or methods defined in one other class or in more than one other class.

xxviii. *Encapsulation:* It is a mechanism that binds together the elements of an abstraction that constitute its structure and behavior.

xxix. *Information hiding:* The process of hiding the structure of an object and the implementation details of its methods.

xxx. *Superclass:* The class from which another subclass inherits its attributes and methods.

xxxi. *Cohesion:* The degree to which the methods within a class are related to one another.

xxxii. *Coupling:* Object A is coupled to object B if and only if A sends a message to B.

xxxiii. *Component Characteristics:* Component characteristics are descriptive attributes of a component that can differentiate it from other components. In this research, the component characteristics of interest are size, churn, complexity, people measures, etc.

xxxiv. *Structural complexity:* This is the complexity resulting from the dependencies and structural properties of the artifact.

xxxv. *Cognitive complexity:* This is the effort or burden on the part of the developer, maintainer, or tester to understand and maintain the system.

xxxvi. *Graph:* A graph according to its mathematical definition is a pair of sets (V, E), where V is a set of vertices (the nodes of the graph), and E is a set of edges, denoting the links between the vertices.

xxxvii. *Directed graph:* A directed graph is a graph where each edge is directed from the first to the second vertex of the pair.

xxxviii. *Release:* A release is normally a new software version where faults were fixed and new functionalities introduced.

xxxix. *Before-release faults:* These are faults or failures observed and reported during the course of development and testing, while

xl. *After-release faults:* These are faults or failures that are observed after the program has been released and shipped to its customers.

xli. *CVS (Concurrent Versioning System):* CVS is a software configuration management (SCM) application that does not store logical changes.

2.3 Software Maintenance

In this section, we explain in-depth what software maintenance is all about. The explanation includes maintenance overview, challenges and categories.

2.3.1 Maintenance Overview

In the field of Software Engineering today, the terms software evolution and software maintenance are often used interchangeably, and this has become a very active research area. Software maintenance is distinguished from software evolution as a post-production and post-deployment activity [3][25]. Software evolution on the other hand, is a stepwise incremental

development of software during its lifetime and it constitutes a critical activity of agile methodology [25].

The criticality of software maintenance stems from the grounds that a substantial amount of efforts is channeledto maintaining existing software systems rather thandeveloping new ones.Companies or organizations have invested huge amounts of cash in their software and are now totallyreliant on these systems. Thus they spend much time and resources maintaining these systems. Software maintenance has been admitted to be the most expensive, labor-intensive and difficult phase in SDLC [2][7][25]. Several studies in the literature have revealed that, in the life of a software system, maintenance alone has been estimated to cost more than the entire development cost [26][27]. In particular, a survey by Erlikh[28], conducted informally in the industryfound that about 85–90% of organizational development costs are maintenance oriented.Other studies conducted by [29][30] suggestabout two-thirds of development costs are maintenance costs. Moreover, the study by [30] revealed that in the entire maintenance cost, more is spent on implementing new requirements than faults fixing.The rough distribution of maintenance costs iscaptured in Figure 2.1.

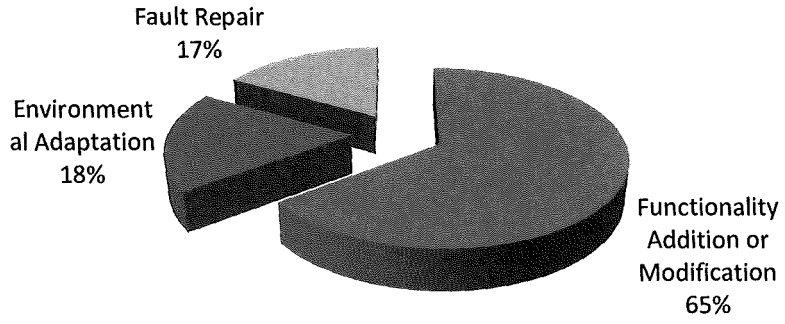


Figure 2.1:Distribution of Software Maintenance Effort [4][30]

However, the exactpercentages are organizational dependent where faults fixing are not the most expensive activity of maintenance [4][30]. As shown in Figure 2.1, it is clear that making the system to adapt to new environments and new or changed functionality is what consumes most maintenance effort. Any solution aimed at improving maintenance productivity would definitely impact software costs. For instance, good Software Engineering methods such as well-defined specification, use of OO development, early fault prediction, more thorough testing can substantially contribute to maintenance cost reduction.Unfortunately, in spite of the progress in software technologies over the last few decades, coupled with several research activities on maintenance methods, the situation has not improved yet. Maintenance cost shows no signs of dropping, rather skyrocketing [7][25]. This increase in cost has seriously affected or limited the capabilities of several maintenance departments to deliver new systems that might be of strategic

importance in their companies [25]. Software maintenance, the last phase of SDLC, is undeniably the most expensive software activity. Software systems unlike other systems are expected to be useful, driven by functionality, flexibility, continuous availability and correct operation [3]. Thus, software maintenance is the activity that ensures a software system exhibits these characteristics in its lifetime.

2.3.2 Maintenance Challenges

Software maintenance is both costly and difficult and therefore poses several challenges. Today, software systems have grown in size and become increasingly complex, in particular, OO software. This means that, when maintaining such systems, a good understanding of the dependencies between the software components is required. However, in practice, the problem is not only on the program comprehension but also the quality or skills of the maintainers, the number of available personnel, and the program's quality. Those who maintain systems are not the original developers of the system and in some cases they have insufficient knowledge of the system or application domain [4]. Consequently, comprehending the rationale behind code poses serious problems to maintainers. Also, as the system undergoes a series of changes or ages, the task of maintaining it turns out to be more complex and more costly than anticipated [3][24]. Other issues that may give rise to maintenance problems are stated in [3] as follows:

i. Maintenance is the most neglected area of software engineering in many organizations, and is usually performed using unplanned techniques;
ii. There is lack of user interest and understanding;
iii. Maintenance personnel are not motivated;
iv. Available program's documentation is insufficient or of poor quality;
v. Source codes are unstructured.

As maintenance is very costly yet very essential, efforts have to be geared towards minimizing its costs. One strategy is a careful understanding of the reasons why the cost is high and then exploration of an alternative technical approach to managing changes during maintenance and evolution.

2.3.3 Maintenance Categories

Software maintenance is a crucial development activity that is aimed at ensuring that software systems carry out their function effectively and efficiently [3]. In this phase of SDLC, only faults fixing and slight adjustments to the SO are thought to take place. However, other activities can also be performed to keep the system useful in all aspects. Three categories of software maintenance exist as stated by IEEE Standard for Software Maintenance [23]:

- *Corrective maintenance:* This is a reactive modification of a software product mainly to correct revealed faults. This type of maintenance includes the correction of errors emanating from code, design, logic and requirements [3],[23]. Among these faults, faults in code are cheaper to correct, faults in design are more costly to repair especially if it involves rewriting of software components, while requirements faults are the most costly since extensive system redesign may be involved [3]. In addition, when corrective maintenance is carried out unscheduled to keep the software system operational, it is known as *emergency maintenance.*
- *Perfective maintenance:* This is the modification of a software product after its delivery to enhance some quality aspects like performance or reliability or to respond to the user's additional or changing need [23][31].
- *Adaptive maintenance:* This type of maintenance involves modification to keep the software system adapted to external environmental changes or useful in changing environments [23].

In the context of this research, only corrective maintenance will be useful in our analysis as it involves faults correction which in turn would be used to predict future faults of OO classes.

2.4 Software Changes

Software changes are inescapable activities during software development and evolution [5]. Change is a indispensable property of any software is necessary to keep a software product continually useful throughout its lifetime. These changes take place in all the phases of SDLC Starting requirements to maintenance [2][5]. To carry out changes on software systems effectively, program comprehension plays a key role. System modification should be taken seriously and the impacts of changes considered because changes in any phase would affect the behaviour of the delivered software product in that phase and in other phases related to it [5]. (See Figure 2.2) Figure 2.2 shows how SO in SDLC relate to one another and the impact thereof. As represented, when software changes made to a software product adversely affect the software components, this may bring inconsistencies to other parts of the original software and the changed software and the impacted components may no longer be consistent with the rest of the software product leading to software deterioration [5]. In other words, as changes are made, many other inconsistencies may surface due to the direct or indirect impacts of the change. Deterioration occurs in many ways because changes to software rarely have the small impact they are believed to have [5]. Therefore, proactively controlled software change is important. All the direct, hidden or indirect impacts resulting from the impacted components must be taken into consideration as to uphold the consistency and integrity of the software.

During the lifetime of a system, the need for software change may originate from a number of reasons such as [3][4]:

- Modification to fix reported defects in the software.
- Change to accommodate modifications in the software system's environment.
- Modification to introduce new requirements or to expand the existing requirements of a system.
- Modification done to prevent malfunctioning of the system.
- Modification to make the processing more efficient.

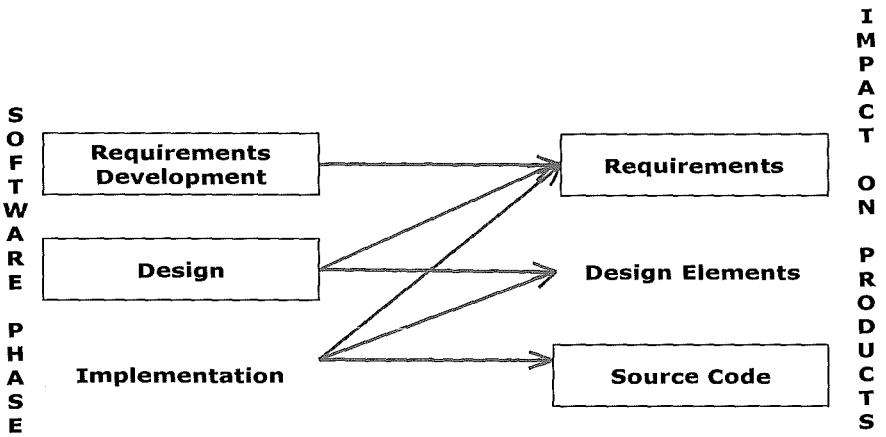


Figure 2.2: Impacts of Change on Software Life-cycle Objects

Irrespective of what initiates software change, CIA is essential in the control of software changes and avoids software deterioration if properly applied. Without CIA, software changes can have unpredictable consequences that could impede the software implementation and onward delivery [2][7].

2.4.1 Change Impact Analysis

As software applications increase in size and complexity, the problems associated with software change also increase accordingly [5]. One such application is OO applications with features such as encapsulation, inheritance, polymorphism, and dynamic binding [7]. Unlike the procedural paradigm, OO features exhibit complex dependencies between classes and attributes that make it difficult to anticipate and identify the ripple-effects of changes [7]. Experience from practitioners has shown that making changes to software with no regard to their impact can lead to "poor effort estimates, delays in release schedules, degraded software design, unreliable software products, and the premature retirement of the software system" [7]. Thus, understanding the proposed change and the impact of the components affected by the proposal are the two essential activities

in software maintenance [5],[7]. This is because regardless of the size of the change, when changes are made they can spread to other parts of the system with enormous impact. In this case, techniques that can assist in comprehending and quantifying the effect of a change to other parts of the software system are indispensable [2]. This is called *software change impact analysis* (CIA).

CIA as defined by Bohner and Arnold [6] is used to comprehend and identify the potential effects caused by changes or estimating what needs to be modified to accomplish a change, through a detailed examination of the consequences of changes in the software. Given a proposed change, the objective is to understand how a proposed change would affect the software components in order to allow more effective prioritization of change requests [3]. An effective CIA can be used before change implementation for program understanding, change impact prediction, and cost estimation [5],[7]. Accordingly, it can be utilized after change implementation to guide regression, select test cases, perform change propagation, and ripple- effect analysis [5]. The cost/benefit ratio can be computed in order to agree on whether to implement the change or not [5]. The processes involved in effective CIA are shown in Figure 2.3.

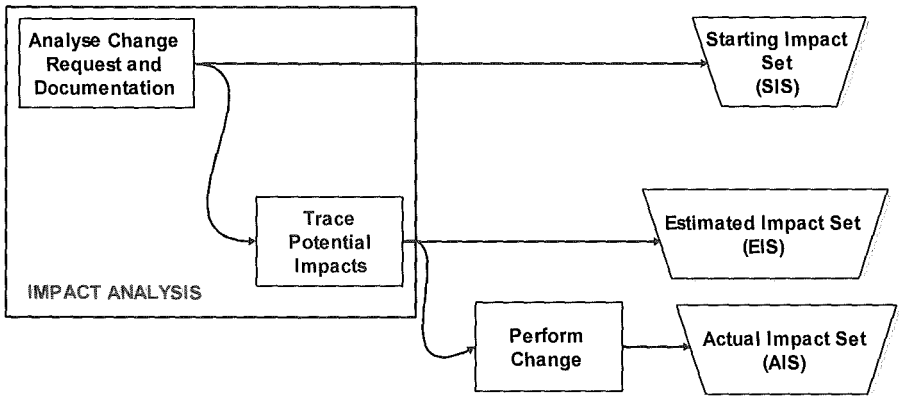


Figure 2.3: Impact Analysis Process [2]

As shown in Figure 2.3, the definition for the notations is presented in Section 2.2.1. SIS constitutes the components set initially thought to be affected by a change while EIS is the set obtained while performing CIA [2][5]. Lastly, AIS constitutes components actually modified. The process needed to achieve these is iterative and discovery in nature [2]. During CIA, in order to identify the impact of a change knowledge of both the dependencies between the software components and how changes spread from one component to another through the dependencies links is important. For OO programs, dependencies between components are captured in objects in terms of relationships such as inheritance, usage, invocations, and so on, while change propagation is expressed in terms of impact rules [5]. Moreover, the impact of a change can be

direct or indirect [5]. In the literature, several studies have expressed the importance of CIA, for example in the issue of Year 2000 (Y2K) [2][7][25].

2.4.2 Impact Analysis Techniques

In today's world, software systems are known to change continuously alongside the requirements and techniques involved. In this case, less-mature systems are gradually made mature. In the field of Software Engineering, the importance of methods, techniques and tool supports have been proved. Thus, for software change to be carried out effectively and efficiently, good techniques are of the essence. Currently, in the perspective of impact analysis, two approaches are used: dependency analysis and traceability analysis [5]. In the work of Chen et al [32], methods of impact analysis are generally classified as *code-based* and *model-based* methods. In particular, *code-based* techniques are centered on utilizing information obtained from the static and dynamic behavior of the program to determine the potential impacts of a change [32]. This means that these methods are based on impacts of a change in the perspective of the source code of the software. Commonly used techniques in performing the task of CIA are the ***static CIA*** technique [12][33] and ***dynamic CIA*** techniques [35][36][37]. These techniques are discussed as follows

2.4.2.1 Static CIA technique

In this CIA technique, all the behaviors and inputs to the program are taken into consideration. Here, the structural dependencies are identified by analyzing both syntactic and semantic dependence of the program [32]. In other words, this technique often performs static analysis on the source code and use the information to construct a static or intermediate representation like dependence graph [7], call graph [33], or software networks which are then used to perform CIA. Several approaches can be utilized to achieve CIA such as depth-first or breadth-first searches. Static CIA have been proved to be conservative and safe but at the expense of precision [32]. The fact is that impact sets produced by static CIA approach are very large and in some cases, problematic for practical use [12][32][33].

2.4.2.2 Dynamic CIA technique

While static CIA involves static code analysis, the dynamic approach consists of program execution, data collection and data analysis [32][35]. This CIA is based on the information collected during the actual execution such as execution traces emanating from methods calls and returns, variable reads or writes, code coverage information, etc. to calculate the impact sets [36]. Unlike static CIA, dynamic CIA techniques are less safe but more precise [36][37]. In addition, the impact sets they generate are small and good for practical use.

Generally, the source *code-based* methods require software developers to know the implementation details of a change request or change implementation plan before time in order to conduct change impact on a software system, otherwise change impacts will be difficult to determine [32]. Moreover, the application of these techniques becomes useful on in the deployment and maintenance phase of SDLC where the program's behavior is available since the product has already been implemented. In the context of this research, the focus is on static CIA as the intention is to develop an approach that will assist in improving the precision.

2.5 Change Process and Software Configuration Management

2.5.1 Change Management

Software change is a fact of life in software development and has to be done following standard processes, and not in a crisscross manner, for effective change control and the enhancement of comprehension on the part of the software engineers. However, the processes involved may vary according to the development processes, the type of application being maintained, and the skills of the developers involved in an organization [3][4]. These processes are captured in Figure 2.4 and could be performed either formally or informally depending on the organizational approach. The processes of change defined in [5][70][71] have the following steps:

1) Change identification and classification
2) Change analysis
3) Accept or reject change
4) Implementation of the change and verification
5) System release

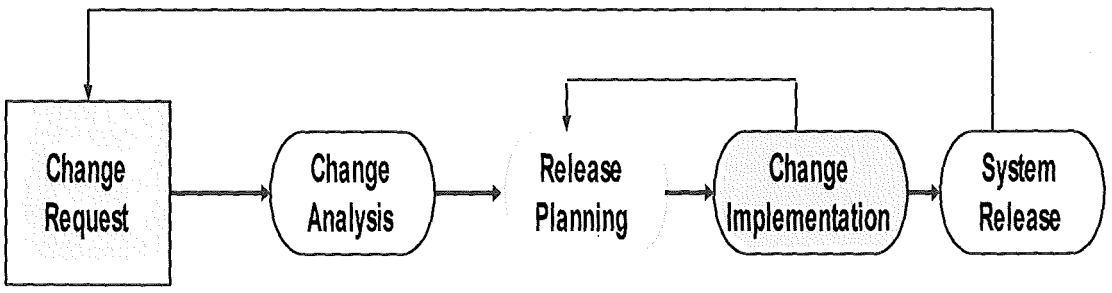


Figure 2.4: Change Process

As presented in Figure 2.4, changes in software systems stem from the *change request* and serve as the process input which may come from existing requirements not implemented in previous releases, new requirements, faults reports from developers or field users, and new ideas for enhancement from the development team [4]. During this step, the change is initiated and

classified accordingly. The classification allows different types of change requests to be dealt with by different relevant bodies as well as obtaining statistics of the distribution of changes over different change sources [70][71]. With this classification, the organization can know whether each change request is minor or major and then assign priorities to them since several change requests may exist at the same time.

After the identification and classification of the various change requests, they are then linked to the components of the system identified to be affected - *change analysis*. The software change is analyzed with respect to the impact on the components affected by the change such as the user interface, functional and non-functional qualities, as well as the cost and schedule [70]. Evaluating the cost and impact of the changes is important as it gives an indication of all components thought to be affected by the change and the cost of implementing them. This analysis is expressed in terms of the needed effort, and the time, budget and resources available to perform the change successfully [5]. However, the key issue remains the identification of all factors impacting the proposed change and the consequences thereof. When all impacts of the change are known and quantified then authority or body (change configuration body (CCB)) responsible can decide on whether to accept or reject the change request based on the cost-benefit analysis of the change [71]. This is the *release planning* stage. At this step, it is deemed important that accurate impact analysis has to be conducted in order for an informed decision about the change request to be made. If the change requests are accepted, a new release of the system is planned and all changes that have been proposed like the fault fix, adaptation, and new functionality are considered [70]. Furthermore, decisions about which changes to implement in the next version of the system are then made. After decisions are made, the changes are implemented and validated, and a new version of the system is released. Change implementation involves modifying documentation, design, and source code as well as carrying out unit testing and verifying the correctness of the change. Once the system is verified to be correct, it is then released to the customers [4]. The process then iterates with a new set of changes proposed for the next release. Thus, the processes are cyclic in nature and continue throughout the system's lifetime.

As we can see in Figure 2.4, following a change process is a valuable step in keeping change manageable. Nevertheless, when changes are not well analyzed or recorded before they are implemented, not reported to the necessary authority or controlled in a way that will advance quality and reduce error, confusion will result [38]. Configuration management is employed during the course of development in order to identify, organize and control modifications to the

software under development [3][4][38]. The goal is to maximize productivity at the expense of low mistakes [38].

2.5.2 Software Configuration Management

Software configuration management is a software quality assurance activity which constitutes an important area of software engineering. As a goal of software engineering, improving the ease with which changes can be accommodated and reducing the expended efforts accruing to the change is what SCM tries to achieve. SCM enables us to achieve a high level of accuracy by avoiding the introduction of new faults while fixing existing faults [3][31]. The activity is to ensure that each software version or release is correct and stable before being released to the customers, and that changes are made accurately, properly and quickly [38].

SCM involves several activities which are defined by [3][38] as:

- *Configuration Identification:* Identifies the software change components that must be kept track of.
- *Configuration Control:* Ensures that component changes are implemented well.
- *Configuration Accounting:* Keeps account or track of what has been changed, when, how, and why.

When changes or modifications to a software system are controlled and managed in this way, it makes the tasks of the maintainers easy and minimizes errors. Keeping track of the details of what has been done to the software makes it possible and easy to know what went wrong and where. For instance, if a bug is reported from the field by the users, it makes it easier to track down the components, the developer responsible etc. In the context of this research, SCM is a critical activity. All the data we need to predict that changes on present component will be risky or that they will be a fault in future components requires the use of faults data and change histories of past releases. Today, SCM is an essential activity of several big software development organizations. Accordingly, several tools exist commercially that can be used in controlling and keeping tracks of changes to software systems. SCM can be done manually or automatically, depending on the tool used and the organization approach.

2.6 Object-Oriented Concepts and Maintenance

OO design approach differs from the structured-oriented approaches primarily due to the different abstraction and real-world modeling concepts that are used. There is a much-admired benefit that applying OO approaches can result in better maintainable and reusable systems [7][8]. This goes with the use of features which are specific to the OO paradigm such as encapsulation, inheritance, polymorphism, dynamic binding and so on [8]. According to [7], OO approaches can yield a

clean, good design that is easier to test, maintain, and extend because the object classes provide a natural unit of modularity. However, in the context of software maintenance, maintainability of a system is a function of its maintenance task's supports, such as impact analysis. Unfortunately, OO program's features have been found to make impact analysis difficult [8]. The complex relationships make it cumbersome to identify the ripple-effects of changes. This is due to the data, control and behavior dependencies that originate from the instance of the class which often affects effective maintenance strategy, a situation called cognitive complexity.

The effects of these features on impact analysis process stem from the fact that, for instance, encapsulation encourages an intended functionality to be achieved by calling many member methods from different classes which means that making a change to one class may affect many classes [8]. On the other hand, inheritance entails that a class can reuse the members of another class. Consequently, fresh dependencies are formed among two classes, so that making a change to one class may affect other classes in association with it [8]. In addition, it is exacerbated by polymorphism where many different implementations of the same specification are performed, while dynamic binding requires the delay of implementation decisions until execution time [8]. This situation is due to the overall structural properties of the software which is a function of poor design approach, programming style and so on. The structural properties are quantified in terms of coupling, cohesion and inheritance.

A. **Encapsulation:** Encapsulation aims at separating implementation of a data object from its specification and controls the way an object's resources are managed, as well as restricting their visibility from others in terms of communication [31][42]. By separating the interface from the implementation, it permits changes without affecting other classes as long as the interface is preserved. Encapsulation is mostly achieved by declaring objects either as being *private*, *public* or *protected,* depending on what needs to be open or hidden from the public view [7]. During the course of performing maintenance tasks on an OO program, encapsulation plays a critical role. Its existence in a program determines the number of components that could be affected by a change. For instance, if a field declared with the keyword *public* is changed, only members from the changed class, subclasses and other classes that reference it will be impacted by the change while for *private* field, only members from the changed class will be impacted. Thus, a good and effective use of encapsulation would make maintenance tasks easy.
B. **Class Cohesion:** Cohesion of a class is a measure of the degree to which the elements of the class belong together. In other words, it is a measure of the relative functional strength of a class which is the closeness of the relationships between the components of the class

[31]. Unlike in the procedural approach, cohesion is usually computed on a per-class or per-object basis in OO software systems [39]. A class is cohesive if the association of elements declared in the class is concentrated on achieving a single functionality to the software system as a whole. In this case, highly cohesive classes having only single responsibility are more desirable than weakly cohesive classes that do many operations and therefore are likely to be less maintainable and reusable [39]. In software maintenance, the class property of cohesion plays an important role in the determination of ripple-effect of a change during CIA. It is a function of a good programming practice. if a class is designed to achieve or implement a single task rather than several tasks, such a class is considered good and maintainable since a change to the class will have no or minimum impact on other classes in the software systems. Thus, developers are encouraged to strive for high cohesion in order to improve the quality of the class and reduce maintenance cost, effort and time.

C. **Class Coupling:** Coupling between classes is the strength or a measure of interdependence among the OO classes in the software system [39]. Coupling between classes is the opposite of cohesion. Unlike cohesion, if **A** and **B** are two different classes, coupling requires that class **A** has to be known in order to understand class **B**, thereby making **A** more closely linked to **B**. That is, classes are said to be highly coupled when they depend on each other classes to the point that a change in one requires changes in other classes that are dependent on it [31]. In this case, changes in one class might affect dependent classes and thus result in ripple effect [39]. Coupling relations has been known to play a critical role in software maintenance and has been proved to be a difficult situation for CIA. This stems from the fact that highly coupled classes are extremely challenging to understand in isolation and reuse because dependent classes must be taken into account. Generally, coupling relations in software systems increases complexity and interface obscurity, decreases encapsulation and possible reuse, and reduces comprehension and maintainability [39]. However, coupling between classes can be reduced by increasing the cohesiveness of the class [31]. To increase the quality of the design and facilitate maintenance, software engineers must design the classes with the goal of high cohesion and low coupling.

D. **Class Inheritance:** Inheritance is a mechanism that permits the definition and implementation of one class based on the definition of existing classes. It is an important feature of the OO paradigm that promotes reuse and allows programmers to define objects incrementally, making use of already existing objects [40]. Inheritance mechanism often

captures "*IS-A*" relation between a super class and its subclass and forms a class hierarchy. The relationship indicates that, an instance of a subclass is also an instance of the superclass, albeit an instance of the subclass is more than that of the superclass. For the existence of inheritance in a class to be useful, it is vital that the hierarchy it forms characterizes a structure present in the application domain and is not formed simply on the basis of reusing some parts of an existing class. Inheritance in different OO programming languages can be broadly classified as strict inheritance and non-strict inheritance [24][31]. Strict inheritance is the easiest form of inheritance where a subclass takes all the properties from the superclass in addition to its properties [31]. In this case, the exact behavior of the superclass is preserved and cannot be modified. On the other hand, non-strict inheritance happens when the subclass does not have all the properties of the superclass or some properties have been redefined [31]. Two forms of non-strict inheritance are subtyping and sub-classing [24].

However, inheritance usage claims to reduce the amount of software maintenance tasks and ease the job of software testers [41]. In the same vein, reusability through inheritance results in systems that are more maintainable, understandable and reliable [40]. There are several empirical studies whose reports contradict the benefits associated with inheritance use in a program. For instance, Harrison et al. [42] in their studies found that a system with inheritance is more difficult to understand and maintain than a system not using inheritance. In another study by [43], it is shown that modification on a system having three levels of inheritance is easier than a system with no inheritance. Several other studies on class inheritance metrics have revealed that the higher the depth of the inheritance tree, the better the reusability of classes, and the more difficult it is to maintain the software system. This means that, in order to facilitate maintenance of software systems with inheritance, the software developers have to keep the inheritance trees as shallow as possible, and remove components reusability through inheritance for ease of program comprehension [41].

2.7 Software Measurement

In every engineering discipline, effective management of processes or products goes with essential ingredients associated with quantification, measurement, and modeling. Software engineering is not an exception. Measurement is defined by Fenton et al [44] as "the process by which numbers are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules". As captured by measurement, entity is an object or event

in the real world, while attribute is the property of an entity [44]. Software measurement has a great influence that enables the current status to be determined and improved. It helps in constructing models, indicators and effective decision-making. Particularly in software engineering, measurement helps engineers to [44]:

i. Characterize and understand what takes place during development and maintenance,
ii. Evaluate product or project status so that they can be controlled,
iii. Assess the effect of technology on software products and processes,
iv. Predict and forecast future performances, and
v. Improve the processes and products based on understanding control.

Measurement in Software Engineering is best expressed in terms of metrics. Software metrics provides a quantitative basis to which a system, a component or process possesses a given attribute [41]. The quest for software metrics stems from the fact that, as a major part of the total development cost of software is dedicated to its maintenance coupled with increases in size and complexity of software products, as well as the consequences of software failure, it is imperative that the quality of the product before and after it is produced is measured. In the context of software maintenance, it is frequently used to provide quantitative information on the software to be maintained [45]. Thus, software metrics play an indispensable role in understanding and controlling software quality since the goal is to produce and deliver a high quality product. It can also be used to estimate project cost and effort, track the progress, evaluate the effectiveness of the software process, and so on [44]. Appropriate use of software metrics can significantly reduce implementation costs and improve the final product's quality, which in turn will reduce maintenance efforts in the future.

In existing literature, several software metrics have been developed or proposed and are broadly classified as product metrics, process metrics, or resources metrics [44].

- ***Product metrics*** quantify or describe the attributes of the products, deliverables and documents such as size, portability, complexity, reliability, design features.
- ***Process metrics*** describe the effectiveness, efficiency and quality of the processes that produce the software product. Examples include time to produce the product, effort required in the process, Number of defects found during testing, etc.
- ***Resource metrics*** describe the project characteristics and execution such as programmer's productivity and skill level, cost and schedule, etc.

2.7.1 Internal and External Attributes

For each entity measured, there are corresponding attributes that characterize the entity and measures that could be used to quantify the attributes. Quality attributes are twofold: *external* and

internal attributes [44]. The difference between these attributes is that, internal attributes are measured purely in terms of the entity itself, separate from its behavior, while and external attributes are only measured in terms of the relationship the entity has with its environment [44].

- ***Internal software product attributes*** can be measured in terms of the product itself such as size, functionality, coupling, cohesion, length, etc. They are the structural properties of the software product which are obtained from a representation of the product, measurable during and after creation of the product [44].
- ***External attributes of products,*** unlike internal attributes can only be measured with respect to how the product relates to its environment. They are concerned with product quality and are properties of the product that are externally visible such as usability, reusability, reliability, fault-proneness and maintainability [44]. These attributes can be measured directly only after the product is created, deployed and has been operational for some time. As a result of this, the focus has been on relating internal attributes to their external qualities.

In the context of software products, these attributes do not exist in isolation. They are related to each other. It is assumed that internal product attributes have a causal impact on external quality. Briand et al. [11][45] provided an expression of the theoretical basis for developing quantitative models that relates OO metrics and external quality attributes. The expression is shown in Figure 2.5 which shows the relationship between OO metrics and fault-proneness due to the effect on cognitive complexity.

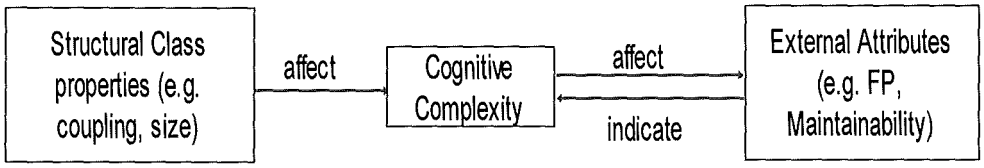


Figure 2.5: Theoretical Bases of OO Product Metrics [45]

Figure 2.5 shows that certain unstable structural properties of classes (e.g. coupling, cohesion, inheritance) have an impact on cognitive complexity which in turn, leads or external quality attributes such as fault-proneness, reduced understandability, and maintainability. Product external attributes can only be obtained late in the development process, but they are inherently relevant to the stakeholders, while the internal attributes are not inherently meaningful, although they can be available early [45]. They are meaningful only when they are seen as indicators of external attributes. Several empirical studies have been performed in the industry and academic environments which conclude that product metrics internal attributes must be associated with important external attributes in order for the metrics to remain useful in the industry.

2.8 Product Metrics

Product metrics are measures of the software product (i.e. internal attributes) at any stage of its development. Several product metrics are available to assess the software product's quality such as the complexity of the software design and the size of the source or object code [34][44]. They are generally classified into traditional and OO metrics [44][46]. The classification is necessary because the need for today's software development lies in the practice of OO techniques. Due to the difference between OO and non-OO techniques, metrics such as cyclomatic complexity and function Points cannot be used to measure the static properties of OO programs [41]. Furthermore, Moreau [47], stressed that traditional metrics cannot be used for OO systems because the computation of the system's complexity in terms of the sum of the complexity of the components is inept for OO systems. Thus, traditional metrics does not measure the structural properties of OO software systems. The different product metrics are discussed in the sections that follow.

2.8.1 Traditional Product Metrics

Traditional metrics constitute measures that quantify the static characteristics of non-OO software systems. Several measures under the traditional metrics have been used for decades in industry as indicators of software quality. These metrics include size measures such as source lines of code (LOC), cyclomatic complexity, function-points, and so on [46]. In this research, we will restrict ourselves to Size and complexity measures.

1) ***Size measures:*** Size measures in the context of a class constitute the measures used in estimating the ease with which software maintainers comprehend source codes to be maintained [46]. Basically, size measures can be computed using different alternatives such as LOC, number of commented and non-commented lines, the number of statements, and the number of blank lines [48]. Of all the size measures, LOC is perhaps the most widely used one that only estimates the volume of code. Though traditional metric are not suitable for OO programs, LOC is the only traditional metrics that can be used in the context of OO program. LOC of a class is a count of all the nonempty and non-commented lines of the body of the class and all of its methods [48]. LOC is an important measure that have been validated empirically as useful predictor of program complexity and programmer performance [52].Counting the LOC is one of the original and simplest approaches to measuring complexity.

2) ***Cyclomatic Complexity:*** This is a measure also known as McCabe, and is used to evaluate the complexity of an algorithm in a method or simply a measure of module control flow

complexity [50]. Cyclomatic complexity has been considered one of the best indicators of system reliability and is the most popular metric among practitioners and researchers [50]. It is based on graph theory which uses control structures to generate a connected graph. The graph represents the control paths through the module and defines complexity of the module in terms of the complexity of the graph [50]. A module is considered good if it has a low value of complexity, otherwise it is complex. Cyclomatic complexity can only be used in procedural programs and cannot be used to measure the class complexity in OO programs due to the existence of inheritance [52]. However, for individual class methods, cyclomatic complexity can be used alongside other measures to assess the complexity of the class. Generally, cyclomatic complexity for a method with a value below ten is recommended, meaning decisions are deferred through message passing.

2.8.2 Object-Oriented Metrics

With the increased popularity of OO paradigm coupled with the complexity its features introduces, there is an increasing need to ensure high software quality since it plays a vital role in any software organization success. This requires the use of OO design metrics to evaluate the quality of software early during software development, since software is intangible. By quantifying the design, its quality will be improved, which in turn could lower the probability of the software being flawed through the revision of improper design [40]. It can be done at considerably smaller cost and reduced efforts than when done later in design or maintenance [40]. OO metrics are measures that are used to capture the quality of the OO software products, particularly the design and code [34][41]. These measures play a critical role in aiding developers to comprehend design aspects of software, and thus improve software quality and developer productivity. OO metrics captures different OO software attributes, such as class complexity, inheritance hierarchy, the internal cohesion and the degree of coupling between different classes [41][45].

Today, the proliferation of OO technology has forced the growth of OO software metrics. In the literature, several OO metrics have been proposed for assessing the quality of OO design and codes for example, by Li and Henry (1993)[14], Chidamber and Kemerer (1994) [41], Abreu and Carapuca (1994) [51], Lorenz and Kidd (1994) [52], Tang et al. (1999) [53], Henderson-Sellers (1996) [54], Cartwright and Shepperd (2000)[55], and Briand et al. (1997) [56]. In addition, several empirical studies have been conducted to validate these metrics. The empirical validation aims at demonstrating how useful the measures are in practice in order to be used in the industry [30]. The overall validity of the measures have been established by building prediction models to show the impact of structural properties of OO measures on external quality attributes like

maintainability and fault-proneness of a class [11][45]. Among the metrics that have been developed so far, the metric suite proposed by Chidamber and Kemerer popularly known as CK metric suit is one of the best-known OO metrics [41]. CK metric suite was among the first to address the important OO design issues with respect to coupling, cohesion, inheritance and class size. The metric suit is one of the most widely empirically validated OO metrics and its use coupled with other complexity measures is increasingly gaining momentum in industry acceptance.

CK metric suite is made of six metrics specific to OO software. A summary of this metrics suite is as follows [41]:

1) ***Weighted methods per class (WMC):*** WMC is a measure of the sum of the complexities of all methods defined in a class. For a package, it is the average of the WMC values of the classes in the package. WMC forms part of the CK suit because it has been proven to be useful in predicting maintenance and testing effort.
2) ***Coupling between Object classes (CBO):*** CBO is the sum of the number of classes that use the member method and/or the instance variables of a given class. In other words, it is the sum of other classes to which a given class is coupled, and thus represents the dependency of one class on other classes in the design.
3) ***Depth of inheritance tree of a class (DIT):*** DIT is a measure of the length of the longest path of inheritance from a given class to the root or superclass in the inheritance hierarchy. Consequently, the deeper the inheritance trees for a class, the tougher the prediction of its behavior might be due to the interaction that exists.
4) ***Number of Children (NOC):*** NOC is a measure of the number of subclasses that inherit directly from the immediate superclass. Reasonable values for NOC show the scope for reuse, otherwise there are considered to be inappropriate abstraction in the design.
5) ***Response for a class (RFC):*** RFC is the number of methods that can potentially be invoked in response to a message received by an object of that class. In this case, the bigger the number of methods that could potentially respond to a message the greater the complexity of that class.
6) ***Lack of Cohesion on Methods (LCOM):*** LCOM is a count of the number of pairs of member methods in the class using no attribute in common, minus the number of pairs of methods that do. If the difference is negative, LCOM is set to zero.

For general design setting, the software developer technically should keep all these metrics at a sound level in order to maintain high quality software. This entails that developers should strive for a low-coupled and highly cohesive design during the early phase of software development.

2.9 Related Works

In this section, we present related works in the area of CIA with respect to OO software systems. Though much research on CIA has been centered on static, dynamic, or a hybrid approach [2][12], Arnold and Bohner are the founders of the field of CIA. Based on their work, several impact analysis techniques or methods have been developed and more are still developing. Today, several approaches exist that utilizes relations involving structural dependencies, dynamic associations and other source code information as well as traceability [6]. In some cases, static program analysis techniques are used, while structural dependencies techniques are used by others [6]. In the perspective of OO software systems, several researches on CIA have been conducted. Li and Offutt [57] studied the effects of encapsulation, inheritance, and, polymorphism on change impact. Their study proposed algorithms for computing the complete effects of changes made in a given class. Nevertheless, changes in the perspective of inheritance and aggregation instances were not fully covered. In another study Anthoiol et al [58], predicted evolving OO systems size starting from the analysis of the classes affected by a change request. Their predicted changes size was based on number of LOC added and modified.

In a similar fashion, in a study by Lee et al [7], OO program entities were represented using program graphs where impact is computed from the static dependencies that exist. In an attempt to keep the ball rolling, Hattori et al [59] developed a tool called Impala. This was an improvement to the work of [7], mainly in reducing false-positives rate. Ryder et al [60] determined the impacts of source code changes using a collection of methods. The CIA technique was chiefly used to prioritize test cases by finding and selecting ones affected and determine the failure position for the changed program. The work was centered on the reactive impact analysis after implementing changes and the result of this technique is a set of potentially affected tests and a set of changes that could have impacted the behavior of the impacted test cases and not OO program components.

In another study Tonella [61] presented a unique representation called the concept lattice of decomposition slices by connecting program variables to the statements, and applying concept analysis in such a setting to aid CIA [61]. The approach was carried out in a straightforward manner, but suits procedural approach better than OO programs. Sharafat and Tahvildari [62] also proposed a probabilistic approach to predict changes in an OO software system using the dependencies obtained from UML diagrams, as well as source code of several releases of a software system using reverse engineering techniques. Abdi et al. [63] proposed the calculation technique of change impact expressions using a meta-model approach to analyze and predict

changes impacts in OO systems. Their CIA approach was evaluated empirically using a real system based on hypothesis testing and they obtained interesting results.

Breech et al. [64] presented coarse-grained impact analysis algorithms that exploit information about how changes can actually propagate due to scoping and parameter passing mechanisms. The study presented influence mechanisms and described both static and dynamic impact analysis algorithms that take advantage of these influence mechanisms. Also, Badri et al. [65] presented a new static technique called CCGImpact for predicting CIA based on control call graphs (CCG) which captures the control related to component calls and generates the different control flow paths in a program. The generated paths in a compacted form are used to identify the potential set of components that may be affected by a given change. Oliveira et al. [66] also presented a hybrid impact analysis technique based on both static and dynamic analysis of OO source code to improve resulting impact estimates in terms of recall. In this study, a prototype was developed to evaluate the approach on a multiple-case quantitative case study. Their results showed a recall improvement of about 90 to 115% for the hybrid, and 21.2% and 39% for the static technique.

In other studies, Chen et al [67] presented an object-based, attribute-oriented approach for software CIA. Their approach handles changes in mixed software product items and produces a holistic impact result for the purpose of total quality management. The study also implemented a prototype to demonstrate their work. In the same way, Wen et al [68] proposed a CIA technique that they based on a compact and effective representation for OOPs, called lattice of class and method dependence (LoCMD). The approach can effectively capture the dependencies between classes and methods. Impact sets were computed based on a metric, impact factor. The study was validated in a case study and the effectiveness of the technique was shown. Xiao-bol et al [69] also presented another form of static CIA technology that was based on program dependence graph (PDG) by analyzing potential change impacts for OO source code. The study supported their work with an implementation of an analysis prototype tool specifically for program in C++ and their results showed that the CIA technique was effective. Jang et al [8] proposed a CIA approach for analyzing change impact in a class hierarchy. The approach was based on the class firewall method that was designed to reduce the regression testing effort significantly. Their study tackled the impacts of changes related to various features of OO program as well as the types of changes occurring at the data, function, class and inheritance level. Lastly,
Sun et al. [12] proposed an approach called OO Class and Member Dependence Graph (OOCMDG) that represented the program to be analyzed based on static CIA. The objective of the study was on the precision improvement of the impact sets depended on the change types and the dependence types between the modified entity and other entities. The impact sets were

computed based on forward and backward walks on OOCMDG according to the impact mechanisms of different change types. Their CIA technique impact sets computed are the fields, methods and classes which are potentially affected by changes.

These are some of the studies or research that has been conducted in the field of CIA on OO programs. However, in all the above highlighted studies, the different CIA approaches on OO programs have been performed on different contexts. These approaches have been designed for change impact prediction and none has incorporated failure prediction, or assessed the risk of a component failing during the course of performing CIA on OO program. The only approaches that are some way similar to ours are [8][12], but on the CIA aspect only. However in this research, our approach is unique and is aimed at merging the two approaches in order to effectively calculate the ripple-effects and reduce or get rid of the risks of software failure during change implementation. By this, we want to predict the fault or failure-proneness of OO classes affected by CIA before implementing a change.

2.10 Chapter Summary

Impact analysis constitutes an important step during the course of maintaining software systems such as quantifying the amount of effort needed to implement a change, helping in the identification of components affected by a change, and prioritizing regression testing. In this chapter, we have discussed software maintenance and impact analysis of OO software systems. In addition, we provided different software measures and discuses some essential previous research done in this study area. The knowledge from the information provided in this chapter will be used or applied in subsequent chapters to achieve the overall goal of this thesis.

CHAPTER 3

Object-Oriented Source Code Change Analysis

3.1 Introduction

This chapter discusses OO concepts in the perspective of impact analysis and presents an overview of the various software entities considered in this thesis. It starts by identifying the components of analysis, their relationship types, properties that govern impact dependency and the various dependency types in an OO program. It then progresses to the categorization of the various code change types applicable to OO systems at the class and members levels. Furthermore, the chapter presents the proposed CIA framework and a program comprehension model that will guide or assist our audience (developers) in carrying out maintenance tasks on OO programs effectively using the CIA technique described in this research.

3.1.1 Change Impact Viewpoints

In OO programming, classes and objects form the basic building blocks in the same way as functions or procedures do for its procedural counterpart. Objects interact with other objects in order to perform some service it provides. This is achieved by message sending which ties the OO program's components together. Thus, objects are encapsulated entities that are composed of: *class data* or *field* and *method*, which are referred to as *members*.

- *Data or field member* defines the *state* of the object, while the
- *Method member* defines the operations on the object's private data.

A class in OO software is composed of these two entities. However, when changes are made to either a member or a class, such change could impact other classes through the connection type that exist between them. In the context of this research, class and its members are at the center of analysis considered at the granularity level.

3.1.2 Basic Component Types

As the basic entity of analysis in this research, a software system is viewed as composed of classes that are connected by different types of links such as *inheritance*, *aggregation*, *association* and so on. A class is made up of *methods* and *field*. Therefore, *class*, *method* and *field* are all entities found in OO program system and are related to one another using any of the links. Each entity is called a *component*. The relationship between the components is captured in Figure 3.1.

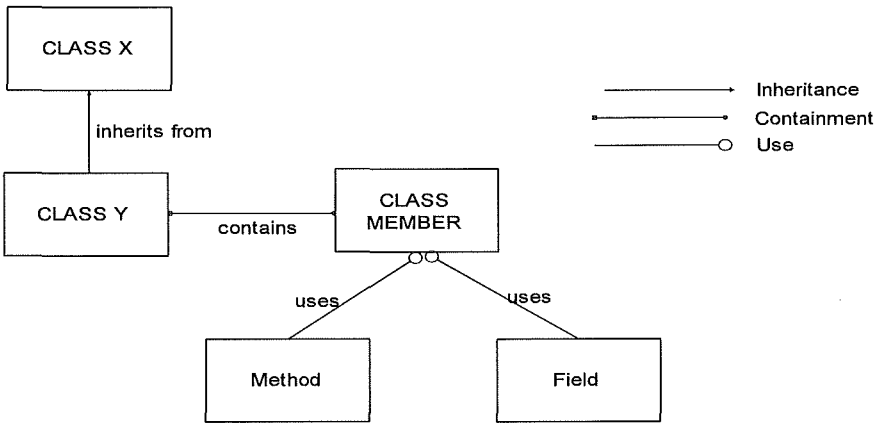


Figure 3.1: Component Relationships

As shown in Figure 3.1, all components have the same probability of being changed. But due to the relationships that exist between them, it is important that when a change is considered on any component, other components that will be affected by the change have to be identified first. This can be achieved through the analysis of the system based on the change proposal and the unaffected components are ignored. The rationale is to ensure that only the components truly affected by the change are modified. This is the task of *impact analysis.*

3.1.3 Relationship Types

OO program classes do not exist in a vacuum but interact with one another through different relationship types that exist between them. The relationships types are of different strengths which indicate the degree of dependency between one component and another. This is critical to impact analysis since changes to one class may propagate to other class or classes. Relationships among components can be either direct or indirect [7][24]. (See Figure. 3.1)

3.1.3.1 Direct relationship

This subsection presents an illustration of the different forms of direct relationships between OO program components. As shown in Figure 3.1, let us consider an OO software system with two classes C_1 and C_2 and the direct relationship $\rightarrow$ between class C_1 and C_2 given by $C_1 \rightarrow C_2$. Thus, $C_1 \rightarrow C_2$ exists if class C_1 and C_2 have either *Containment*, or *Usage*, or *Invocation* or *Inheritance relationship.*

1. *Containment Relationship:*

   The containment relationship denotes class C_1 ***aggregates*** or ***contains*** class C_2. This type of relationship exists if C_2 is declared as a class member of C_1. By implication, it indicates that definition of a class C_1 implies objects of another class C_2, thereby suggesting some kind of a whole–part relationship between C_1 and C_2.

2. *Use Relationship:*

   Usage relationships denote class C_1 ***uses*** class C_2. The relationship type can formed from different perspectives. For instance, it exists if either class C_1 ***contains*** class C_2 or C1 sends a message to C_2.

3. *Inheritance Relationship:*

   In this type of relationship, if class C_1 is declared as a superclass of C_2 or vice versa then it is said class C_2 ***inherits*** class C_1 or vice versa. In this case, C_1 *or* C_2 can inherit the properties of C_2 *or* C_1 such as instance variables, interfaces, and instance methods as though they were defined within class C_1*or* C_2.

4. *Invocation Relationship:*

   Let two member methods be *m* and *n*. Therefore, if *m* of class C_1 ***invokes*** or calls method *n* of class C_2, we referred to it as invocation relationship. That is, if m_{c1} calls some services n_{c2}.

3.1.3.2 Indirect Relationship

Indirect relationship in OO program is simply a cyclic relationship in which an impact of one class on another indirectly affects itself. For example, we say C_1 has an indirect relationship with C_2 if a path of the form C_{21}, C_{22}..., C_{2n}, such that $C_1 \rightarrow C_{21}$, $C_{22} \rightarrow C_{23}$, ... , $C_{2n} \rightarrow C_2$, expressed by $C_1 \rightarrow^+ C_2$ [24].

3.1.4 Impact Dependency Properties

Classes must interact with one another to achieve the overall system's objectives. However, the presence of complex dependencies among OO software components may lead to direct or indirect effects on other components when changes are made. Properties that determine how one component affects others based on their relationships are the *reflexive*, *transitive* and *cyclic* property [7][24]. We define these properties as follows. Let C_1, C_2 and C_3 be components of a system and the symbol, $\leftrightarrow$ the change impact dependency. Therefore:

i. ***Reflexivity:*** A class C_1 is said to be *reflexive* if C_1 depends on itself, given by $C_1 \leftrightarrow C_1$. The property of this type denotes that a change to class C_1 will affect itself in terms of its members. In an implementation, we called this a *local* or *membership* dependency.

ii. ***Transitivity:*** A change impact dependency is called *transitivity* if a change to one class may propagate to other classes indirectly. That is, if C_1 impacts C_2 and C_2 impacts C_3, then by implication C_1 impacts C_3. It is represented as $C_1 \leftrightarrow C_2$, $C_2 \leftrightarrow C_3 \Rightarrow C_1 \rightarrow C_3$.

iii. ***Cyclic:*** This is the cyclic nature property and it signifies that cycles exist in the impact dependency representation. In other words, a cyclic property exists if there is a path of

relationships that returns back to the components where an impact started such as $C_1 \rightarrow \rightarrow C_2 \rightarrow C_1$. Thus, a change to a class affects itself through other classes.

3.2 Object-Oriented System Dependencies

Dependencies can be described as the kind of relationships that exist in source codes that are technical in nature. It is the kind of relationship between two components such that changes to one may have an effect that will require other components to be changed. In the context of a program, they can be described as semantic or syntactic relationships between the program components representing various aspects of the program's control and data flow. They reflect the potential for one component to impact or be impacted by the system's elements, including other components [79]. In the context of this thesis and based on the maintainer's point of view, the understanding of dependencies is based on [79]. In this case, a dependency is considered as a form of *direct relationship* that is defined as follows:

Definition 3.1: *[Class Dependency]*
Let C_1 and C_2 be classes in a software system and whenever C_2 uses another C_1, then we say, C_2 depends on C_1 if and only if changes to C_1 have a direct effect on the behavior of C_2. That is, C2.y() uses C1.e(). In this case, C_2 is called the dependant class and the C_1 is called the dependency. (See Fig. 3.2)

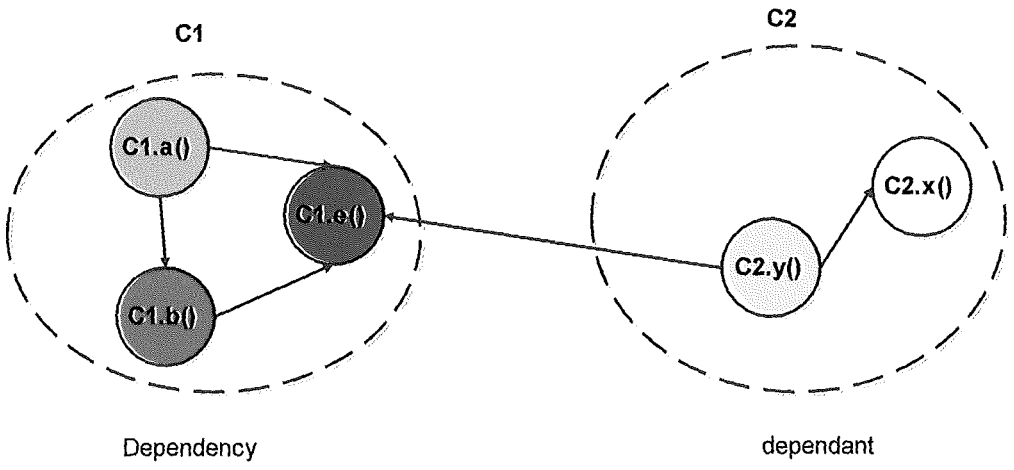


Figure 3.2: Class Dependency

As shown in Figure 3.2, C_1 and C_2 are the classes while C1.a(), C1.b(), C1.e(), C2.x() and C2.y() are the methods in the classes. C_2 uses or depends on C_1 and the representation indicates that a maintainer who will modify C_1 must take the possible side effects in C_2 into account before the change can be implemented. As shown in Figure 3.2, dependencies between OO classes are basically the result of software decomposition as well as composition which is thus essential to

manage software complexity. As a relationship, the degree of the dependency is the most important aspect as it indicates to what extent one component depends on the other.

3.2.1 Effects of Dependencies on Maintenance

There are several types of dependencies that exist in OO software which are broadly categorized into *structural* and *logical* dependencies. The dependencies are due to OO program features and have been identified as the factor that makes it difficult to predict or identify the ripple-effects of changes in OO software systems [7]. For instance, the lesser and simpler the connections between components, the easier it is to understand each component in isolation. Thus, reducing relationships between components also reduces the paths along which changes and faults can propagate into other components of the system, thereby removing unpredictable ripple-effects. Accordingly, dependencies also reduce the reusability of software components, especially highly dependent or coupled components. The complex dependencies are based on the nature of collaborations between object components such as *usage*, *invocation* or *call*, *inheritance*, *implementation* and *membership* [12][69]. (See Fig. 3.3)

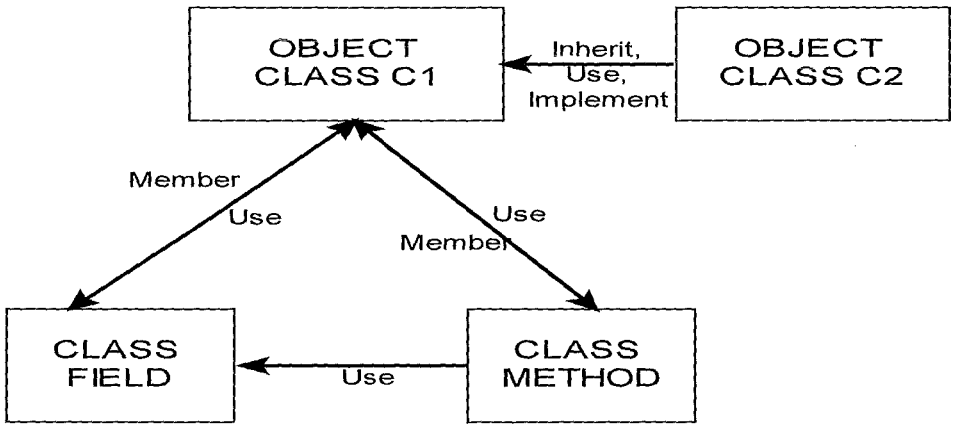


Figure 3.3: OO Components Dependencies

Since much of the complexity is known to originate from the relationships among object components, it is therefore necessary to identify all the key dependencies in order to perform impact analysis effectively. The dependencies: *usage*, *inheritance*, *membership* and *invocation* will be used in this thesis. Detailed descriptions of the dependencies are discussed in the section that follows.

3.2.2 Dependencies Types

The different types of dependencies we considered in this study are discussed in this section as follows:

3.2.2.1 Class Dependencies Type

Dependencies between classes can exist in different forms: *inheritance*, *implementation* and the *usage* dependencies. Implementation dependency is dealt with as inheritance dependency.

1. ***Inheritance dependencies (H)***

   Given two classes C_1 and C_2, *Inheritance* exists if:

   - C_2 inherits from C_1
   - C_1 inherits from C_3
   - C_2 indirectly inherits from C_3

2. ***Usage Dependencies (U)***

   The *usage* dependency denotes that an object of the class may be declared as the *field* of another class or a situation where the class is instantiated and used in another *method*. That is,

   i. C_1 uses C_2

   ii. C_1 aggregates or contains C_2

   iii. C_1 aggregates or contains C_2 by value or reference

3.2.2.2 Class-Member Dependencies Type

This type of dependency is mainly between the class and its member where *methods* and *fields* are members of the class. In addition, class can be instantiated or used within a method. In that case, the dependency is the *usage* (U) type. Generally, this is called the *Membership* or *local* dependency and it exists in the following ways:

1. Class and Method

i. Method ***m*** returns object of C_1

ii. C_1 implements method ***m***

2. Class and Field (f)

i. ***f*** is an instance of C_1

ii. ***f*** is a class variable of C_1

iii. ***f*** is an instance variable of C_1

iv. ***f*** is defined by C_1

3.2.2.3 Member Method-Field Dependencies Type

In OO software system, methods often *use* the member field(s). The dependency is then the *usage* type. In the context of this research, the nature of *usage* dependency between *f* and *m* can take one of the following forms:

i. ***f*** is a parameter for method ***m***

ii. ***f*** is a local variable in method ***m***

iii. ***f*** is a non-local variable used in ***m***

iv. f is defined by m

3.2.2.4 Member Method Dependences Type

Member methods are regularly invoked in an OO program to carry out some operations. Thus, dependencies between these methods are presented as either a *call* or *invocation*. The nature of method invocations are as follows:

i. Method m_1 calls method m_2

ii. Method m_1 overrides m_2

3.2.2.5 Dependencies between Fields

The dependency between class member fields is typically the *usage* dependency. For instance, f_1 of method m_1 defines or uses f_2 of method m_2.

3.3 Change Type Categorization

There are different types of code changes that can be made to an OO program and different changes have different impacts diffusion [12][69]. Consequently, it is essential to carefully identify the type of changes that needs to be made and their impacts when faced with a change proposal. This is because knowing these change types and their impact range is fundamental to software maintenance.

Based on the level of granularity considered in this research, changes to OO program components can be considered at three different levels: *field*, *method*, *class* and *package*. In the context of this thesis, software systems written in a Java programming language are considered, though most of the changes can still be applied to other OO languages such as C++. The different types of components changes considered in this thesis are based on the work of [12][24][63][69] and are based on any of the three change kinds: *deletion*, *addition* and *modification*.

3.3.1 Class Change Types Category

A class can be changed during maintenance by any of the following operations:

i. Addition of a new empty class,

ii. Deletion of a whole class,

iii. Modification to class usage modifiers,

iv. Inheritance derivation between classes.

For usage modifiers (i.e. *access*, *static*, *abstract*, and *final modifiers*), details are as follows:

- *Access modifier:* The access modifier of a class is one that imposes a restriction on the access to the class members. That is, the information hiding principle which stems from the use of keywords such as *public*, *protected* and *private*.

- *Abstract Modifiers:* The abstract modifier in an existing class demonstrates that a class is incomplete and composed of an abstract method(s), thus cannot be instantiated.
- *Final modifiers:* The final modifier denotes that no class can inherit any feature from the final class. The sole purpose is to prevent the class from being inherited. Thus, a class can be declared final if its definition is complete and no derived classes are desired or required.
- *Static modifiers:* The modifier static pertains only to member classes and is often used in an inner class. When adding the static modifier to an inner class, the inner class becomes a nested top-level class and can be used as a normal top-level class [12].

Furthermore, change to the inheritance relation between two classes is not an exception. Such changes include the deletion, addition of a parent class (or an interface) and the inheritance derivation between two classes. The different types of changes embedded on a class are shown in Table 3.1.

3.3.2 Class Methods Change Types Category

A method of a class can also be changed in different ways like the class itself. The categories of changes that are relevant at the method level are:

- Adding an ordinary new method,
- Deleting a method, and
- Modifying an existing method.

Modification to an existing method can also take the following forms: modifying method usage modifiers, method name, method parameters, return type of a method and statements within the method body [12][24]. With the exception of method modifiers, other method modifications are simply referred to as *method signature change*. Changes to methods usage modifiers of a method are similar to that of a class but differ in level of operation.

- *Access modifiers:* The method access modifier specifies the level of visibility of the method outside the class to which it belongs. Just like in a class, access modifiers are also declared with the keywords; *private*, *protected* and *public*.
- *Abstract modifiers:* A method is said to be an abstract method if it is declared without an implementation. Its intent is to be overridden in every class that inherits from the base class. In addition, if abstract methods are declared in a class, the class itself must be declared *abstract*.
- *Final modifiers:* A method final modifier indicates that the method cannot be overridden by inherited classes. That is, it disallows the inherited classes to change the meaning of the method.

- *Static modifier:* A static modifier on a method definition specifies that this method belongs only to the class and not a particular instance of the class. This implies that it can be called using the class name and not the object name.

Table 3.2 presents the different types of changes that can be performed at the class method level.

Table 3.1: Class Change Types

Change *(chtype)*	**Change Description**	
CA	Class Addition (common)	
CD	Class Deletion	
ICA	Increase Class Access	
DCA	Decrease Class Access	
AAbC	Add "Abstract" Modifier to the Class	
DAbC	Delete "Abstract" Modifier from the Class	
AFC	Add Final Modifier to the Class	
DFC	Delete Final Modifier to the Class	
ASC	Add Static Modifier to the Class	
DSC	Delete Static Modifier to the Class	
CID	Class Inheritance Derivation	
	CID1	Private ->Public
	CID2	Public ->Private
CCN	Change Class Name	
APC	Add Parent Class	
DPC	Delete Parent Class	
MPC	Modify the Parent of the Class	

** Common class=> non-abstract, static, or final class

Table 3.2: Class Method Change Types

Change *(chtype)*	**Change Description**
AM	Add ordinary Method
DM	Delete existing Method
IMA	Increase Method Access
DMA	Decrease Method Access
CSM	Change Statement within a Method
AAbM	Add Abstract modifier to a Method
DAbM	Delete Abstract modifier from a Method
MRM	Modify Return Type of a Method
MNPM	Modify Name of Parameters of the Method
MPM	Modify Parameters of the Method
MNM	Modify Name of the Method
AFM	Add Final modifier to the Method
DFM	Delete Final modifier from the Method
ASM	Add Static modifier to the Method
DSM	Delete Static modifier from the Method

3.3.3 Class Field Change Types Category

They category of changes that can occur at the class field level also includes the addition, deletion and modification to a field. Like the class and class method, the modification to a class field

involves the field usage modifier (i.e. *final*, *static* and *access*), type and name. The usage modifiers are discussed below to distinguish them from that of the class or method.

- *Access modifiers:* Access modifier to a field is similar to the description of the method's access modifier.
- *Final modifier:* The final modifier of a field denotes that the field cannot have its value changed after its initialization. In such case, the field is a constant.
- *Static modifier:* A class field where only one instance of the field exists, in spite of the number of class instances created. Here, access is via the class name other than an object and the static variables can be used by all objects of the class to perform some joint actions.

In addition, other changes that are applicable at the class field level are the field type and the field name change. Table 3.3 presents all the different types of code changes at the member field level.

3.3.4 Package Change Types Category

Like the class and its members, the categories of changes that can occur at the package level also include the addition, deletion and modification. We captured these changes in Table 3.4.

Table 3.3: Class Field Change Types

Change *(chtype)*	**Change Description**
AF	Add ordinary Field
DF	Delete ordinary Field
IFA	Increase Field Access
DFA	Decrease Field Access
MFT	Modify Field Type
MFN	Modify Field Name
AFF	Add Final modifier to Field
DFF	Delete Final modifier from Field
ASF	Add Static modifier to Field
DSF	Delete Static modifier from Field

Table 3.4: Package Change Types

Change*(chtype)*	**Change Description**
APk	Add a new Package
DPk	Delete existing Package
MPkN	Modify the name of an existing Package

3.4 Impact Model

An impact model is a model that is specifically designed for CIA. It is used to predict which components of a software system will be impacted if a change were to be implemented. In this research, the CIA technique will be based on *static* CIA technique. Static CIA involves the computation of impact sets based on the static information from the source code. In this case, given a Java program *P*, the task will be to perform static analysis on the source code and extract components as well as their dependencies based on the change type considered. In order to speed up the process of CIA, we employed complex software network representation where classes, methods and fields are the *nodes* and the dependencies between them are the *edges* in the network. Although, static models are less precise, and generate large impact sets which are sometimes difficult for practical use, they are known to safe [12]. Despite this pitfall, the intent in this research is to ensure that the precision of the technique is improved by reducing the impact set to the minimum. Thus, the objective is to formulate an approach that will ensure that the computed impact set as small as possible. As the focal point is on static and syntactic impacts, emphasis will be placed on impacts that are dependent on the static nature of the source code. Impacts stemming from the execution of a program (i.e. dynamic impacts) such as polymorphism and dynamic binding are not considered in this research.

3.5 Impact Analysis Framework

3.5.1 Motivation

In the last decade, several software firms have witnessed and incorporated OO technology into their software development environments. Today, this technology has gained worldwide popularity either in small, medium or large software organizations. In particular, OO programs are now the mainstream of developing software systems, and thus there is a need to understand the problems of maintaining such systems, together with comprehension strategies to enhance the maintenance task. This is important because maintenance tasks today are mostly carried out by persons who are unfamiliar with the OO software system. Hence, they have to understand the processes underlying this activity before maintenance can be carried out successfully. However, the cognitive complexity associated with an OO software system has a great impact on its maintenance. Having high cognitive complexity in a system can lead to the system's components exhibiting unacceptable external qualities, such as increased fault-proneness, reduced maintainability and understandability [11][42].

Today, several models and program comprehension strategies in terms of direction and breadth [74] exist to neutralize the effect of OO software cognitive complexity. However, they are not

generic for all situations. To this end, we have proposed a framework that will assist in carrying out CIA and a generic cognitive model for understanding OO programs during the course of performing maintenance task.

3.5.2 Targeted Audience

Several organizations have invested huge sums of money on their software and are now completely dependent on them. They consider their systems critical business assets and they have to invest in system change to maintain the value of these assets. This change does not happen automatically, but is performed by trained personnel. Often, the people may not be familiar with the systems because they were not the developers. In such events, they have to employ the underlying mechanisms of program comprehension in order to carry out maintenance on the unfamiliar system. We understand that undergraduate students are the future developers of software development firms. Most of them are going to be involved in maintaining OO programs when they start working as graduates. Thus, it is imperative that they understand the approach to use and the comprehension strategies they will apply while maintaining such systems. This forms the basis of dedicating this research to improving maintenance to undergraduate students.

3.5.3 The CIA Framework

As an indispensable property of any software, changes are made to realize various change proposals of software systems. Based on the change proposal, the task of the maintainer is to analyze and evaluate the system in order to effectively predict the impacts of the change. Since However, it has been revealed that about 85-90% of the total development cost of a system is expended on maintenance [28]. Moreover, OO software components have complex dependencies that often time adversely impact maintenance and their components, classes in particular, are not exempted from being faulty. Hence, it is vital that during CIA and before actual changes are implemented, change impact prediction be performed along affected components' fault-proneness prediction. This is necessary to ensure that the risks and cost of the change implementation are reduced to the minimum.

3.5.3.1 Framework Description

This framework is designed to serve as a guide to maintainers when carrying out maintenance tasks of making changes to OO software. The design is unique and incorporates both *change impact prediction* and *failure prediction* which depends on the size of the system. The framework is captured in Figure 3.4 and the key activities are dependency analysis and extraction, CIA process, early failure prediction and change recommendation. Figure 3.4 presents an overview of the proposed CIA framework which is the focal point of this research.

A. **Dependency Analysis and Extraction:** This is the first stage which is aimed at facilitating OO program comprehension and effective CIA. On the proposed framework, the original OO source code has to be analyzed by constructing an intermediate source code representation (IR). The IR is simple and clearly reveals all the possible components (classes, methods and fields), and their dependencies (inheritance, membership, invocation and usage) [12] as well as permitting the quantification of the overall program complexity. This is to provide a good understanding of how components relate to one another and to facilitate CIA activities in the next stage. The representation is based on *complex software networks*. The goal in this stage is to assist maintainers to:

   1) Visualize the structure and dependencies of the system,
   2) Compute the degree of components' coupling,
   3) Predict the impact of a change, and
   4) Quantify the risk propagation of each component with respect to fault in small sized systems.

   Perhaps these activities will enable software maintainers to take appropriate actions during the course of CIA in the later stage.

B. **Impact prediction:** After the construction of the IR, the next and crucial task is to perform the actual CIA. The essence is to help the maintainer quantify or determine which OO software components in the original software systems will be affected by the change proposal or which will bring inconsistencies to the software if changes are made. With the IR, this stage ensures that the impacts of changes are localized as possible. Based on the nature of OO software, we have proposed a technique called *impact diffusion* which will be used to precisely predict the impact of changes. Thus, the *impact diffusion* is based on three influential factors:

   1. The type change performed on the object components,
   2. The dependency types that connect one component to another, and
   3. The behavior and impact diffusion range of each change and dependency type.

   The rationale is that, in OO program unlike non-OO program, the effect of changes are dependent on the change type performed and the nature of the dependencies between the components affected by the changes [12]. These determinant factors are to be taken into account in order to precisely predict the effect of a change and to allow decisions to be taken as early as possible on whether to implement or reject a change. The goal is to improve the accuracy and precision of the predicted *impact set* which is the output of the stage.

C. **Early failure prediction:** With the predicted impact set at hand, the goal in this stage is to assess affected components for fault-proneness or which of them may lead to failure when changes are made. The motivation is that, with CIA process shown in Figure 2 of Chapter 2, it is quite clear that the process is specifically used to predict the impact of changes while components' fault-proneness is not taken into account before the actual changes are made. As OO software components are not fault or failure, the position of this research is that, if changes not meant to fix existing components' faults are made, they could create some undesirable effects or increase the likelihood of the software to fail. Predicting faults early would allow mitigating actions to be focused on the high risks components or take alternative actions before changes are implemented. This prediction will be based on probability and the process will be based on the size of the system. Since OO software systems are of different sizes: small, medium and large, the following approaches can be applied when maintaining such systems.

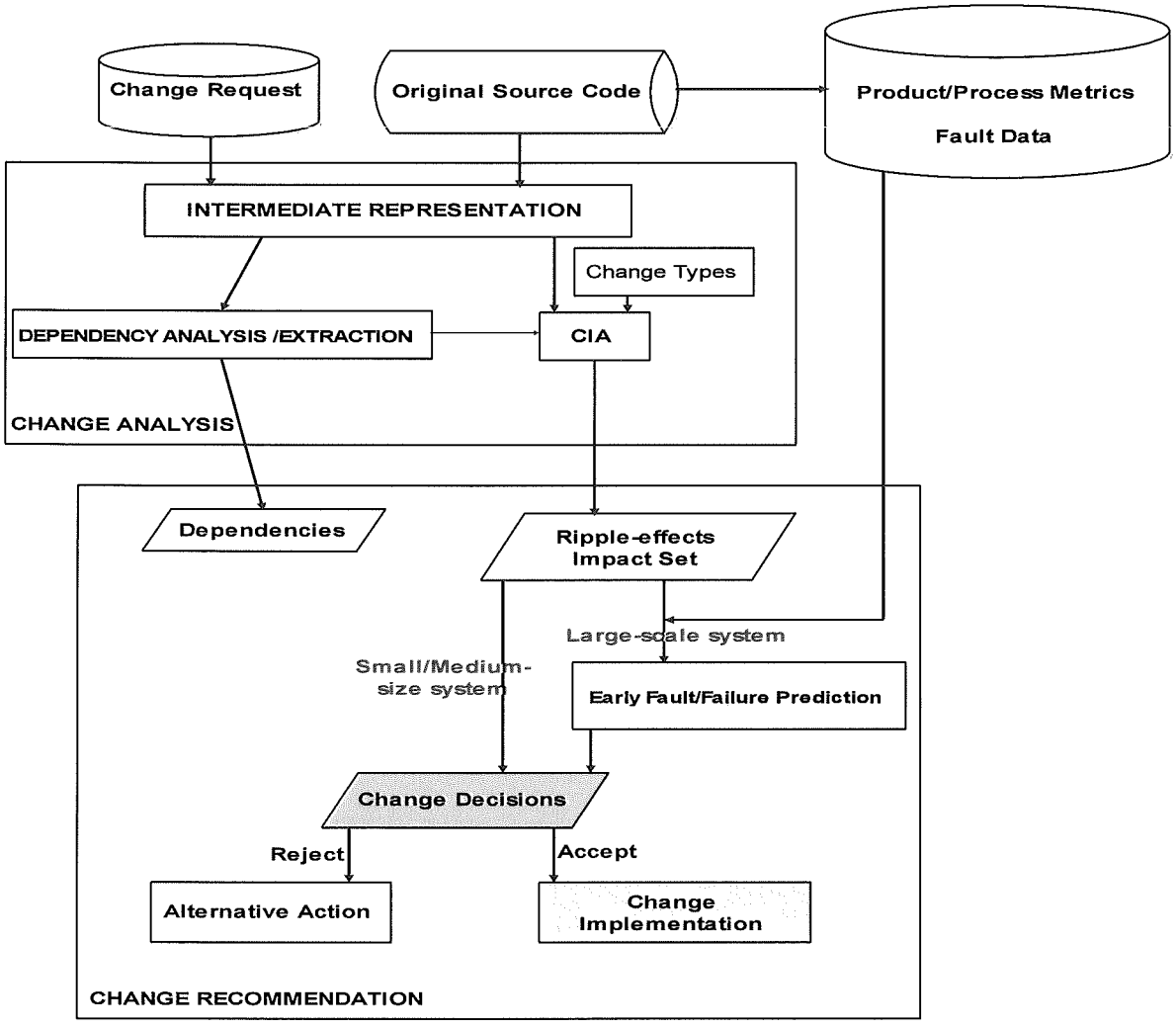


Figure 3.4: Proposed CIA Frameworks

- ***Small/medium sized systems:*** For small or medium sized systems, the quality of the components identified as impact set can be assessed by computing the probability of fault propagation using their dependencies in the complex software networks. To this end, the risks components pose to other components they connect to are computed and the value obtained is used to take decisions during change implementation. Based on the computed values, the higher the probability the higher would the risk of the fault propagation be. In the same vein, a smaller risk value would signify that the presence of a fault in such component poses no serious impact to other components and modification can be performed hitch-free. The knowledge of the risks based on the values obtained will assist the software maintainer to take extra precautions during the course of implementing the actual changes.
- ***Large-scale systems:*** In the perspective of large OO software systems, using the complex software networks might not be appropriate. In this case, the quality of the systems can be assessed via pure prediction using software metrics such as code metrics, past change and fault histories as well as suitable fault prediction model. Several empirical studies in the literature have confirmed the relationship between product and process metrics and fault-proneness [18]. To carry out the prediction, all the measures extracted from either the previous or current version of the software stored in the database will be used to predict whether a component affected by a change will be faulty or not. The motivation is that software quality is known to play a crucial role in the success and failure of any software organization. However, in large software systems, providing high quality in development has been deemed a complex and a laborious activity [18][105]. In this case, it is important that the available resources are focused on the most critical parts of the system to ensure customers' satisfaction. That is to say, the early identification of faulty components before changes are made is of importance both for reduction of maintenance efforts, costs and risks while preserving software quality. This will in turn facilitate software testing and inspection activities.

D. Change implementation: After identifying the impact set and their overall quality assessed, the next step is to take decisions on whether to implement the change or not. In other words, this is the acceptance or rejection stage. Deciding on whether to implement a change or not is important because, for example, if a change proposal is known to trigger significant undesirable effects over the entire system and majority of the affected classes are fault-prone, one decision could be to reject the change or to consider an additional change plan or redesign the system through strategies like refactoring, or accept the

change proposal. A change is only implemented if the impact and the risks are known to be small or after validation and verification activities have been performed on the affected faulty parts. Otherwise, it is rejected if it is known to have deteriorating effects on the whole system. The essence is also to reduce the cost of risky changes.

3.6 Program Comprehension

Understanding is fundamental to an effective change process. This stems from the fact that modifying software, perhaps, an OO program, without understanding the dependencies that exist may result in some unpredictable effects on its quality or increase the risk of failure in the field. Thus, before implementing any change, it is crucial to understand the software product as a whole as well as the parts of it and the components that will be impacted by the particular change. This requires that the maintainer must know the *WHA*T, *WHERE* and *HOW* of what he or she is about to do. In other words, a maintainer needs to have the knowledge of WHAT the intended system does, see WHERE the changes are to be made, and know HOW the affected parts that need modifications work.

Program comprehension is, therefore, the process used by individual developers or maintainers to understand "unfamiliar" program codes [72]. It is a central activity and a determinant factor of the success or failure of program maintenance. Unfortunately, it has been recognized as a critical cognitive and a complex problem solving activity [73]. Albeit, it has generally been accepted that understanding a program consumes a substantial amount of maintenance time, effort and resources [3], understanding the principal techniques of OO program comprehension is thus essential for performing maintenance effectively as it is known to involve a great deal of cognitive complexity. During the course of program understanding, it is necessary that maintainers construct their own mental model of the program. The cognitive processes alongside other information structures used to develop the mental model are portrayed by a cognitive model. Thus, a program comprehension strategy is a method that is used to form a mental model of the OO program. In literature, there are several strategies that are used by developers to form a mental model of a program: the *direction* and *breadth* of comprehension [74] though differs in cognitive structures and cognitive processes.

3.6.1 Cognitive Model

In context of this thesis, the cognitive model we have proposed for the targeted audience is the mixed model which is a combination of both the *direction* and *breadth* of comprehension. For the *direction*, we used the *opportunistic model* by Letovsky [75], and for the *breadth*, we used the *as-needed strategy* by Littman et al [74]. The model is intended to guide the maintainers in building

a mental model of an OO program that will assist in understanding the program for onward maintenance tasks. The model is captured in Figure 3.5.

3.6.1.1 Opportunistic Model

Comprehension with this model centers on three key and balancing elements: *knowledge base*, a *mental model* and an *assimilation process*. (See Figure 3.5) The model is called opportunistic because it is believed that developers or maintainers tend to comprehend in an opportunistic way, switching between top-down and bottom-up strategies flexibly [74]. In the context of this research, the intuition is that this model will help maintainers to build a mental model that will assist them in understanding and performing OO software maintenance effectively. The model elements are discussed as follows:

A. Knowledge Base: The knowledge base is aligned with our intuition based on the maintainer's expertise and background knowledge utilized in the understanding task. The intuitive belief is that, before a maintainer can quickly and effectively understand an OO program for onward maintenance, he or she should have a good knowledge of the OO paradigm that ranges from concepts to implementation strategies. This requires that the maintainer should know and be able to map change request to OO source code, identify different dependencies that exist in the target source program, different taxonomy of change made to OO software components, the impact diffusion of each change type and the overall structural complexity of the target OO program in terms of fault-proneness and maintainability.

B. Mental Model: The mental model articulates the maintainer's understand of the OO program at hand. Based on the knowledge base, the maintainer is expected to construct an abstract representation of the program, by creating a picture of the packages, if present, the classes, methods and fields as well as the interconnection between them. In other words, the maintainer has to create a picture of the overall structure of the system. although, the information created might not be complete, it has to communicate key information about the OO system. This is important because having a clue of the structure and complexity of the program will allow attention to be focused on where attention is due for maintenance to succeed.

C. Assimilation Process: Assimilation process expresses the method used to acquire information from the source code or change request. It operates in an opportunistic manner because it enables software maintainer s to proceed (either in a top-down or bottom-up approach) in whichever way they assume yields the highest return in knowledge gain. In this case, classes, methods, fields, and packages will be used as beacons.

i. *Top-down Approach:* Based on this approach, a maintainer can gain knowledge or an understanding of the system at hand by first identifying the number of packages, the number of classes in each package, the different methods and the different fields found in each class or methods.

ii. *Bottom-up Approach:* In this approach, the maintainer could gain an understanding of the program by first identifying the different fields and methods in each class, the different classes in each package, and the number of packages in the target system.

This information will assist the maintainer to invoke the correct plans, taking the knowledge base into account to form a mental model of the program to be understood.

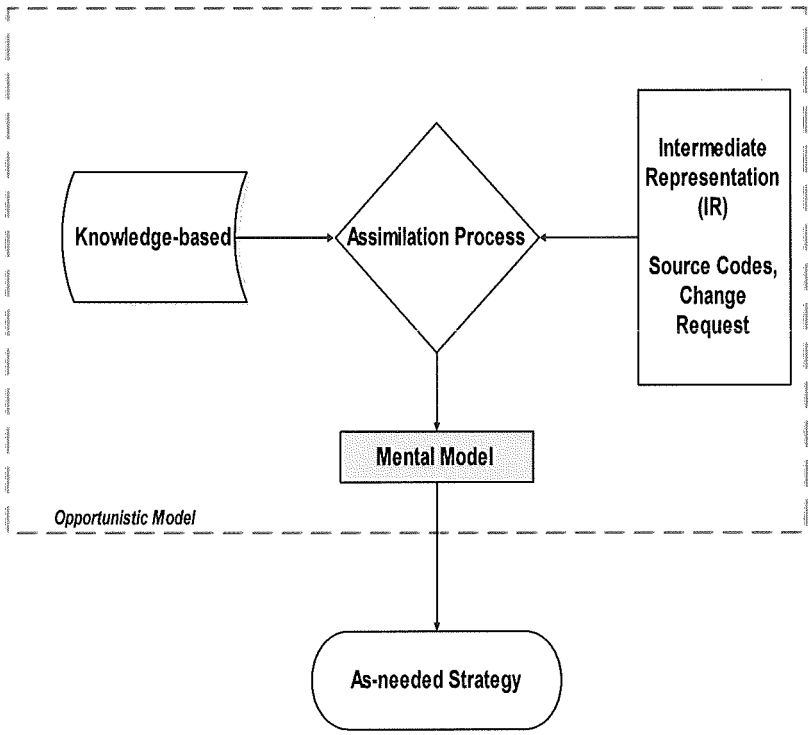


Figure 3.5 Opportunistic-as-needed Comprehension Model

3.6.1.2 As-needed strategy model

This strategy alongside the opportunistic model will assist the software maintainer to carry out maintenance tasks effectively and efficiently. Moreover, it can be applied to both medium-sized and large programs. An *As-needed strategy* will enable the maintainer to concentrate on the areas of the source code thought to affect or be affected by a change. This is necessary in order to gain a static in-depth knowledge of the system to successfully carry out the modification task. For an OO program in the context of this research, the knowledge base of the opportunistic model such as change types, different impact diffusion mechanisms, and so on are indispensable.

All these concepts and models in this chapter will be used or applied in subsequent chapters in order to achieve the objectives and overall goal underlying this thesis.

3.7 Chapter Summary

In this chapter, we have defined and discussed the basic components of analysis in this research as well as CIA frameworks and the comprehension model. OO program class and its members are all components at the granular level that is important to our analysis in the perspective of impact analysis. The relationship that exists among components and the identification of the dependencies they form as well as the accrued change type is important for OO program impact analysis. Furthermore, we also discussed the proposed CIA framework and program comprehension model that will assist developers in carrying out the task of maintenance effectively with respect to the audience. The models are intended to guide them in developing a mental model used in understanding the part of the OO source code they have to modify. This model will be used alongside the technique of CIA that will be discussed in subsequent chapters to perform changes on source code effectively and efficiently.

NWU LIBRARY

CHAPTER 4

Intermediate Source Code Representations

4.1 Introduction

This chapter deals with component dependency analysis of OO program code and the construction of an IR thereof, as a tool for program comprehension during maintenance. It begins with a high-level overview of OO software CIA with a form of program representation, and then progresses to a low-level or detailed IR with respect to change and fault propagation probability using the concept of *complex software networks*. The goal is to develop approaches for representing OO software systems that will assist software maintainers to improve comprehension, evaluate the risks and impacts of change as well as the quality of the software under maintenance.

4.1.1 Motivation

With the increasing quest to solve complex real-world problems using large-scale software systems, software has grown in size and complexities. Consequently, it becomes difficult to understand its underlying properties and its quality becomes unmanageable [76][78]. In a nut shell, all these factors have been known to affect the understandability and maintainability of OO program. Therefore, during the maintenance, one way to contain this problem and enhance comprehension is to construct an IR of the software systems using *complex software networks*. This is possible because OO programs share global network characteristics such as small world and scale free [77]. They have clear structures and components such as:

i. Fields
ii. Methods
iii. Classes
iv. Packages

In addition, these components and their dependencies are responsive to extraction and analysis. In the context of this thesis, the goal of using a complex network is to proffer an effective representation of an OO program that makes explicit, its implicit and complex structure.

4.2 Complex Networks in Software Systems

Complex networks in recent decades have gained increasing momentum and this includes software systems because of their topological structure [76][77]. A software system can be modeled as a complex network where software components are represented as nodes and their

interactions are edges. In particular, OO program structures are quantified by their structural properties in terms of components and the relationships which are critical to their quality. In the same vein, the quality of the software structure is inversely proportional to fault-proneness and maintainability. In this case, the better the structure of the system, the better the maintainability, the lower the fault-proneness and the lower the risk, cost, time and effort required will be. Since OO systems are the mainstream of software development today, changes during the maintenance stage are inevitable. Due to complex interdependencies between OO program components, the change or fault in one part often requires changes or faults of several other parts in a way not anticipated. Hence, the objectives are:

1) It would be desirable for software maintainers to ensure that a particular change does not involve a wide range of components. This requires static analysis of codes, representation and comprehension of OO components and their relationships in order to discover which parts will be truly impacted and examine them for further impacts.
2) With the increasing complexity of OO software systems, a fault in one component can propagate to several other components directly or indirectly connected to it. Thus, it would also be desirable to quantitatively analyze the quality of the entire structure of complex software networks. This approach will allow degree of risk of a fault in a component on other components, either directly or indirectly through different dependencies, to be measured. Analyzing software structure quantitatively would help the software maintainer to know beforehand the risk of fault propagation in the components, and take mitigating actions where necessary, in order to reduce the risk of software failure.

We give a detailed explanation of the above objectives in the subsequent sections.

4.3 Dependency Analysis and Extraction

Dependency analysis is a critical activity and an integral part of CIA framework used in understanding how one entity relates to another through effective representation of the original software. In the realm of OO system maintenance, a system dependency graph (SDG) is one such approach where relationships and components in source codes are statically extracted and analyzed at very fine level of granularity [12]. Nevertheless, constructing this representation requires much precautions and good knowledge of OO program design structure. This is because wrong representation could lead to over or under-estimation of *impact sets*. Since the *static* CIA technique is used to predict the *impact sets* of changes made to a Java program ***P***, we therefore model ***P*** using the idea of *software complex networks* called *OO Component Dependency Network*, OOComDN hereafter.

4.3.1 Data Collection

In this thesis, data collection is referred to as the process involved in extracting the various components of the software and their dependencies. In this case, data will be the statically collected manually from Java program code. In the same vein, we considered different kinds of dependencies: the dependence between fields and methods, dependencies between methods, dependencies between classes, and dependencies between classes and methods. (See chapter 3, Section of 3.2). These dependencies are commonly categorized as *MEMBERSHIP*, *INHERITANCE*, *USAGE*, and *INVOCATION*.

4.3.2 Object-Oriented Component Dependency Network

OOComDN is used to represent components and their relationships in OO software systems. On OOComDN, the OO software components are nodes and the interaction between every pair of the components if any, is a directed weighted edge with an edge type indicating the probability that a *change* or *fault* in one component may propagate to the other. The network representation will be considered in two perspectives: *change* and *risk diffusion* networks.

4.3.2.1 Change Diffusion Network

In change diffusion network (CDN), the OO software system is represented using a weighted direction graph, G, where components are the vertices and the dependencies among the components are the edges. CDN is used to represent the software components and their relationships for onward maintenance tasks, perhaps CIA. It represents the structure of the OO source code and assists software maintainers in quantifying which components will be truly affected by a change. In other words, the representation is basically used to discover the evolution mechanism of the software system. Based on CDN, the following definitions are important:

Definition 4.1: *[Dependency Types (D^{Type})]*

Interaction between every pair of the components is inevitable in OO systems and once a change is considered on a specific component, it may propagate to other parts of the system. In this case, it is related to the changed component via a link. We called this link the $\boldsymbol{D^{type}}$.

As discussed in Section 3.2 of Chapter 3, four dependency types are essential: inheritance (**H**), usage (**U**), invocation (**V**), and membership (**M**). These dependencies are non-numeric weight assigned to the edges in OOComDN.

Definition 4.2: *[OOComDN -1]*

In OOComDN-1, the nodes represent components (fields, methods, classes and packages) of a specific OO software system, while the edges represent the dependencies. Each component is

represented by only one node and the weighted-directed edge between two nodes indicates one component either uses, member, invokes or inherits the others.

Thus, OOComDN can be described as:

$$\text{OOComDN -1} = < (\mathbf{N}, \boldsymbol{D}^{E}), \boldsymbol{D}^{type} >$$

Where $\mathbf{N} = \mathbf{N}^{Pk} + \mathbf{N}^{C} + \mathbf{N}^{M} + \mathbf{N}^{F}$ and $\mathbf{D}^{E} = \mathbf{N}\times\mathbf{N}\times \boldsymbol{D}^{type}$ represents the set of various edges with dependencies types, $\boldsymbol{D}^{type}$. We referred to $\boldsymbol{D}^{type}$ as the weight of the graph and $\mathbf{N}^{Pk}$, $\mathbf{N}^{C}$, $\mathbf{N}^{M}$ and $\mathbf{N}^{F}$ represent the set of packages, classes, member methods and fields respectively. (See Figure 4.3)

With the OOComDN using Figure 3.2 of chapter 3 for instance, let C_1 and C_2 be two components in P. If component C_2 uses or inherits or invokes C_1, there will be an edge emanating from node C_2 to node C_1. In this case, the multiplicities of these dependencies are taken into account based on the *type of change* to be performed on a given node in the network. The weight of each directed edge will indicate the probability that a change in node C_2 will affect class C_1. A typical illustration of the OOComDN is shown in Figure 4.3, using Java program example of Figure 4.1. The various shapes used for representing each component in OOComDN are shown on Table 4.1.

The IR is achieved by first constructing OOComDN-1 as shown in Figure 4.3 based on the original software, the Java program, **P** .The construction is quite simple since it does not analyze deeply into the method body such as control statements, return types, and so on. The approach is almost equivalent to that of [12][77], and all the implicit dependencies covers are clearly revealed. In addition, OOComDN-1 is not complex and the components involved are countable. However, $\boldsymbol{D}^{type}$ as one of the determinant factors of identifying impact propagation have to be well-known. Although, its identification may involve more tasks during the course of CIA, the IR is good and is practicable for impact analysis of OO software systems.

Table 4.1 OOComDN Features

Component	Shape
Package	Doted square
Class	Oval
Method	Rectangle
Field	Circle

4.3.1.2 Software Network Degree

The degree of a node in a network or graph is the number of edges the node has against other nodes connected to it. To identify the nature of coupling of each component on the network and the structural complexity of the software, the computation of *degree* of each node in terms of *in-degree* and the *out-degree* on the OOComDN-1 is required. The rationale for the computation is to give an insight into how components are related to one another in terms of coupling. The computation is done at the class level and it requires pruning of the OOComDN-1 to classes and their dependency types as shown in Table 4.2. The definition is as follows:

Definition 4.3:*[Degree of OOComDN-1]*

Given, OOComDN, $< (N, D^E), D^{type} >$*, with an adjacency matrix* A_{ij}*, the degree of a vertex,* Z_i*, We defined the out-degree of a node as the number of edges or connections originating from that node. It is given by* $|Z^{out}(n_i)|$ *which is the sum of the* i^{th} *column of the* A_{ji}*.*

$$Z^{out} = \sum A_{ji} \quad \ldots\ldots 4.1$$

While in-degree of a node, n_i *is the total number of edges or connections onto that node and is given by* $|Z^{in}(n_i)|$ *which is the sum of the* i^{th} *row of the* A_{ij}*.*

$$Z^{in} = \sum A_{ij} \quad \ldots\ldots 4.2$$

$Z^{tot}(n_i)$ *is the total number of directed edges into and out of the node,* $n_i \in N$*. It is simply the sum of* Z^{in} *and* Z^{out}*.*

$$Z^{tot} = Z^{in} + Z^{out} \quad \ldots\ldots 4.3$$

In other words, $Z^{in}(n_i)$ indicates the number of classes that have dependency on class $n_i \in N$ and $Z^{out}(n_i)$ the number of classes on which class $n_i \in N$ depends. The complex relationships that exist among several software components in an OO program always increase the structural complexity of the software systems. As shown in Table 1, for instance, class A has one Z^{in} for the ordered paired (B,A) and (D,A) and no Z^{out}. In addition, Z^{tot} is a measure of the overall complexity of the program. This shows the nature of coupling in A which will assist a maintainer in identifying the complexity of the classes prior to performing CIA. As the complex relationships among OO software components often lead to structural complexity of the software system as well as cognitive complexity, being similar to Chidamber-Kemerer's (CK) Coupling between Object Classes (CBO) metric [41], Z in the software networks would show the degree to which each class depends on other classes. Thus, we used Z to measure the degree of coupling in a small or medium sized system.

Table 4.2 In-degree and Out-degree for OOComDG of Figure 4.2

Node, n_i	Z^{in}	Z^{out}	Z^{tot}
A	2	-	2
B	2	1	3
C	1	1	2
D	3	-	3

```
package p1;
public class A {

public A(){};
private int d;

public void M1()
{ d=2; }

public int M2(int x)
{ M1();
x= d + 10;
return x; }}

public class B extends A
{
public B() {};
private int a;

public void M3()
{ a=5; }

public int M4(int b)
{ M3();
int c = a+b+10;
return c;
}}
```

```
package p2;
import p1.*;

public class C {
    public C(){};
    private p1.B k;

public void M5()
{   k.M4();  }}

class D extends C {
public D() {};

private String q;

public void M6()
{ q="Boy!";
B j ;  j.M4();
A p;  p.M1();
}}
```

Figure 4.1: Sample Java Program

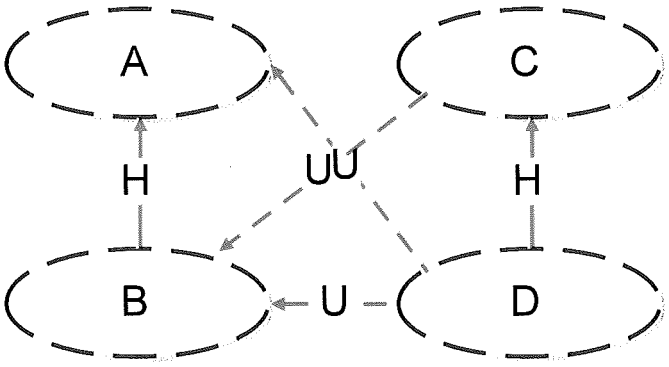


Figure 4.2: Class level OOComDN-1 for Figure 4.1

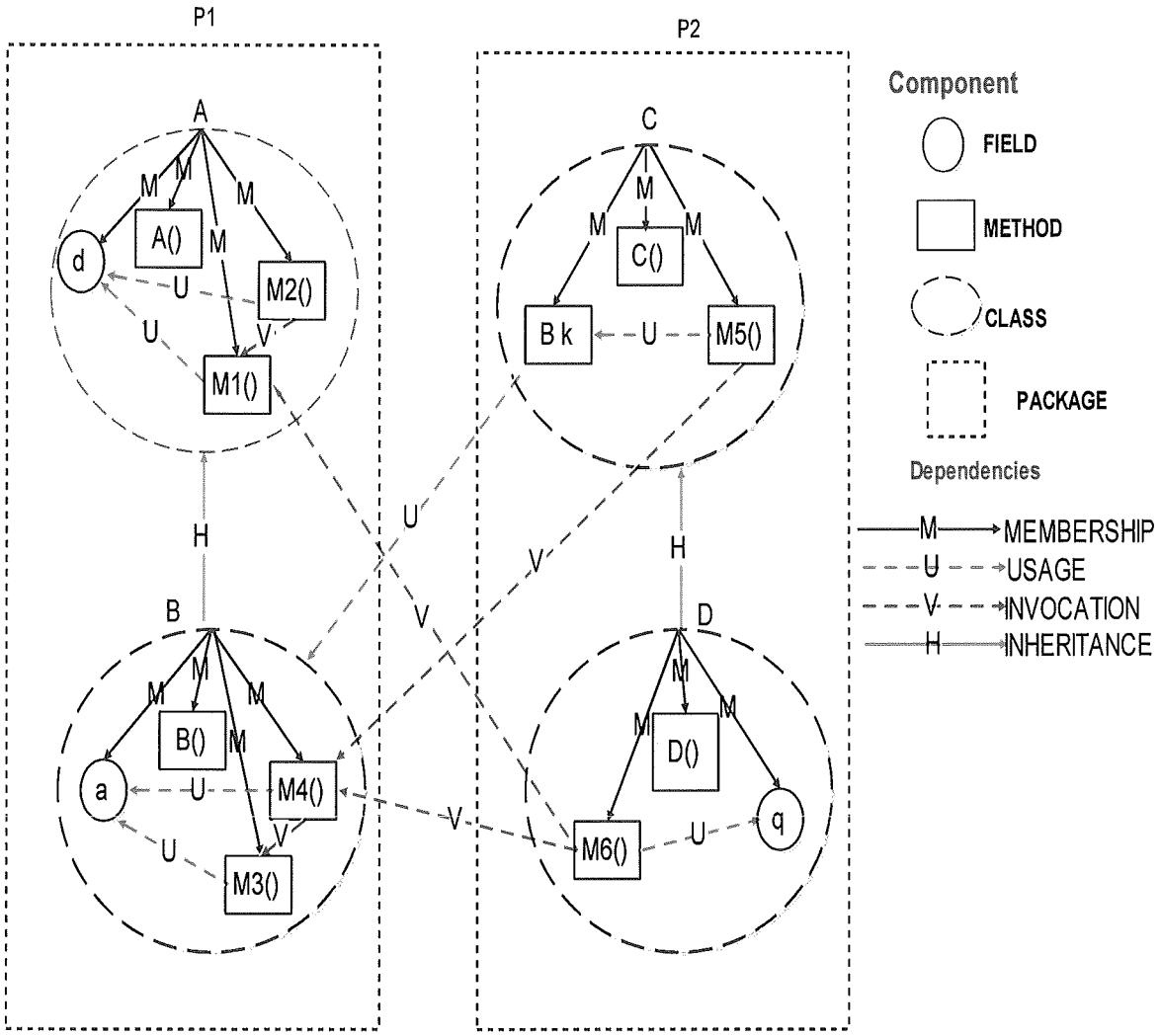


Figure 4.3: Member-Class level OOComDN for Figure 4.1

4.3.1.3 Fault Diffusion Network

Fault diffusion network (FDN) represented just as CDN. The only difference is that the semantics of the relationship is neglected and every relationship has the same importance. FDN is used to characterize the risks a component poses on others due to the direct or indirect dependency existing between them. The justification is that, though it is believed that a fault in one component will propagate to other components that depend on it, the case is not always true with respect to OO software systems [77]. The intuition is that, OO program class is composed of several fields and methods and a class is considered faulty if it has at least one fault emanating from either itself or its members. In this case, members of another class that depends on such faulty class do not all connect to the faulty member directly or indirectly. Hence, the propagation of fault from one component to another is based on probability. In this case, we adopt the approach proposed by [77]. The definition is stated as follows:

Definition 4.4: *[OOComDN-2]*

In FDN the nodes represent the classes and a class is represented by only one node in the entire OOComDN-2. Interactions between classes are represented by directed edges.

Thus, OOComDN-2 can be described as:

$$\text{OOComDN -2} = \langle N_C, D_C, P_b \rangle$$

In the definition 4.4, $\mathbf{N_C}$ is the set of classes, $\mathbf{D_C}$ is the set of edges linking one class to another and $\mathbf{P_b}$ is the probability that a fault in a class will propagate to another. The interaction is based on the principle that, if members in class, say **D** use class members of **A**, **B**, an edge will originate from the node of the member in class **D** to the node in **A, B** and vice versa. For simplicity, in FDN, only the existence of dependency is considered while the $\mathbf{D^{Type}}$ is ignored. Additionally, the multiplicity of the dependencies regardless of how many times a class depend on another class is ignored. The numerical weight on each $\mathbf{D_C}$ in a class is the same for all members which represents the probability that a fault in a class will impact or spread to other class members they relate to. (See Figure 4.4)

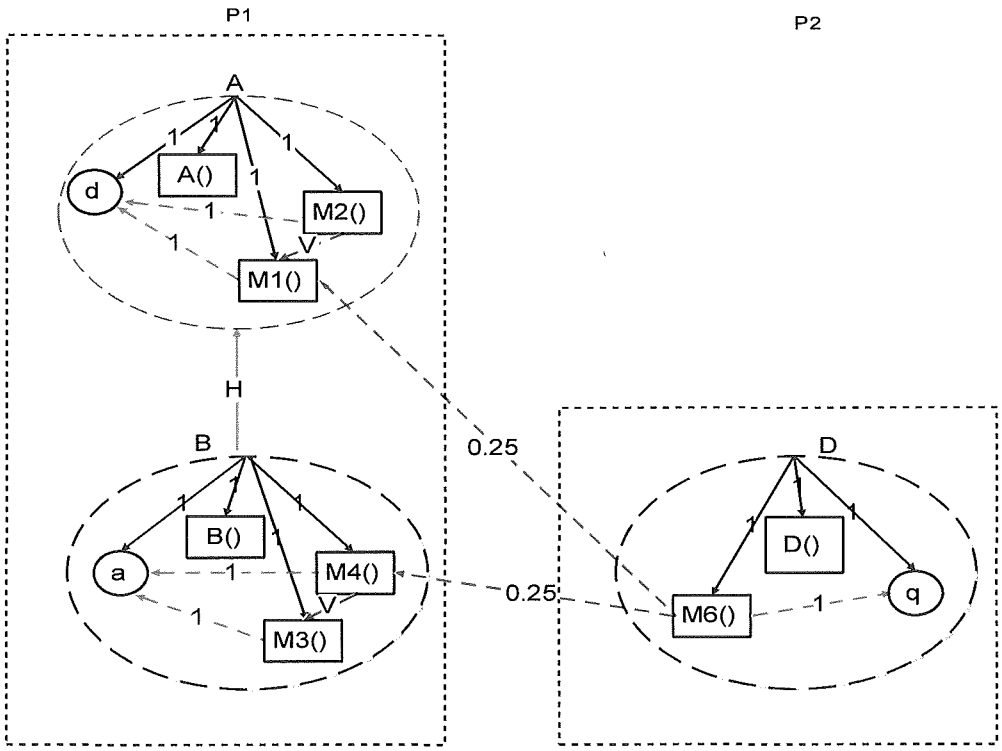


Figure 4.4: Class Fault Propagation Probability

Definition 4.5: *[Fault Propagation Probability]*

*Let **P** be a java program having class **i** and class **j**, where class **j** depends on class **i**. We therefore define the probability of fault propagating from class **i** to class **j** as **P_b (i,j)**. This is defined as follows:*

$$\mathbf{P_b(j,i)} = \frac{|\mathbf{CM(i,j)}|}{|\mathbf{MT_j}|} \quad \text{.....4.4}$$

According to [77], **CM(i,j)** is the set of members in class ***j*** whose faults will propagate to the members in class ***i***, which they are directly or indirectly linked to, thereby rendering the class faulty, and $\mathbf{MT_j}$ is the total number of class members present in class ***j***.

CM(D,A) = {*M1()*} and $\mathbf{MT_A}$ = {*d, A(), M1(), M2()*}

CM(D,B) = {*M4()*} and $\mathbf{MT_B}$ = {*a, B(), M3(), M4()*}

Figure 4.4 captured the fault propagation probability in a class. The edges of all members in a class are denoted by 1. It indicates the probability that a member of the class will be faulty due to the dependency it has with a faulty member. That is, every member of a class has the same probability of being faulty if a member they depend on is faulty. However, for inter-class dependency, the case is not always true. Each class has its own probability value which is based on the number of members in that class that depends on the faulty class. For instance, as shown in Figure 4.4, it is clear that class **D** depends on class **A** and **B** as follows:

(D.M6(),A) = {M1()} = D.M6() → A.M1()

(D.M6(),B) = {M4()} =D.M6() → B.M4()

Therefore,

$$\mathbf{P_b(D,A)} = \frac{|\mathbf{M1()}|}{|\{d,M1(),M2(),A()\}|} = \frac{1}{4} = \mathbf{0.25 = 25\%}$$

$$\mathbf{P_b(D,BA)} = \frac{|\mathbf{M4()}|}{|\{a,M3(),M4(),B()\}|} = \frac{1}{4} = \mathbf{0.25 = 25\%}$$

This computation shows that $\mathbf{P_b}$ **(D, A) =** $\mathbf{P_b}$ **(D, B) = 25%**. This denotes that, since M6() in class **D** depend on class **A** and **B**, the probability that a fault in class **A** or **B** will transmit faults to class **D** is only 25%. For inheritance dependency type, the probability will not be computed because members in the classes are not linked directly. We based the computation on the fact that, the higher the probability, the higher the risk of the fault propagation would be. Additionally, a smaller risk value signifies that a fault in the measured component poses no serious impact on the other components and modification can be performed hitch-free. This idea stemmed from the fact that, if a class in which other classes depend on is faulty and is not detected before a change not meant to fix it is made, there is the probability that the faults may propagate to other components

connected to it. To avoid such problem, it is important that during CIA, the risks propagation probability of all the affected classes identified as *impact set* should be computed before actual changes are made. The approach will assist the maintainer to quantitatively measure the structural quality of the software through the assessment of the potential risks. The essence is to allow the maintainer know which components affected by a change proposal will have a higher risk probability of transmitting faults to its neighbors during changes. It will in turn allow mitigating actions to be focused on those high risk components in time to avoid the cost of software failure.

4.4 Dependency Matrix

This section discusses the strategies for identifying initial *impact set* of a change during CIA on the OOComDN-1. It is based on *adjacency matrix* representation. The objective is to provide a *high-level identification* of the relationship between the classes or members of the original program. That is, the designed will assist in the identification of the **SIS** with respect to the change proposal. The correct identification of **SIS** is crucial to the correct computation of the **EIS** which is geared towards improvement of the overall precision. The strategy involves the transformation of the OOComDN-1 into three separate dependency matrices:

i. Class dependency matrix (CDM),
ii. Intra-membership relation matrix (MRM), and
iii. Inter-membership relation matrix (IRM).

With these matrices, CDM is a high-level matrix that is extracted from the high-level structure of the entire system that is only composed of classes and their dependencies (See Figure 4.2), while MRM and IRM are extracts of the CDM which involve class members' dependencies. The two words dependency and relation are used in the matrices to denote class-to-class relationship and member-to-member relationship respectively. Each matrix is explained as follows.

4.4.1 Class Dependency Matrix

CDM is an adjacency matrix representing both dependencies and relations among different classes of OO program. Based on the different source code change type of OO programs, it is obvious that changes are not only limited to class members but also to other classes and package where a link exist. Thus, CDM is used for the basis of class changes. The following definition is given:

Definition 4.6: *[Class dependency matrix (CDM)]*
Given the ***OOComDN-1****, two classes* ***A, B*** $\in$ ***N*** *for instance, we define CDM as follows:*

$$CDM = [m_{ij}] = \begin{cases} -=1 & if\ there\ exist\ intra-class\ dependency, A \in N \\ +=1 & if\ there\ exist\ inter-class\ dependency,\ A \to B \nvdash A, B \in N \\ 0 & Otherwise \end{cases} \quad ...4.5$$

The definition captured in equation (4.5) is three-fold and it indicates the following:

i. $\mathbf{M_{ij}}$ = "-"=**1** value denotes that there is a local or internal relationship within a class and its members, "-" $\in$ $\mathbf{D^E}$. The significance is that a change to a class affects the class itself. We called "-" the intra-dependency value.

ii. $\mathbf{M_{ij}}$ = "+"=1 as long as **i** $\neq$j, indicates that there is external dependency "+" $\in$ $\mathbf{D^E}$ for **A** $\rightarrow$**B**. We called "+" the inter-dependency value, indicating that a change to class **B** will affect **A** and any other classes related to it. And lastly,

iii. $\mathbf{M_{ij}}$ = **0** value denotes that there is no dependency between class A and B. (see Table 4.3)

Table 4.3: Class Dependency Matrix for Figure 4.2

n_i/n_j	A	B	C	D
A	-/+/0	-/+/0	-/+/0	-/+/0
B	-/+/0	-/+/0	-/+/0	-/+/0
C	-/+/0	-/+/0	-/+/0	-/+/0
D	-/+/0	-/+/0	-/+/0	-/+/0

In Table 4.3, each matrix value shows implicitly the $\boldsymbol{D^{type}}$ (i.e. *usage, invocation and inheritance)* where the directed edges direction is from the column class to the row class.

4.4.2 Intra and Inter-membership Relation Matrix

These two matrices are used to represent the relationship between members of a class and members of other classes connected to it respectively. To understand these matrices, the following formal definitions are stated:

Definition 4.7: *[Intra- member Relation Matrix]*

Given the OOComDN-1, a class, say $A = \{a_1, a_2, \ldots, a_m\} \in N$ *and a dependency "-"* $\in N$*, we then define the intra- member relation matrix for dependency "-" of class A as follows:*

$$K_A = [k_{ij}] = \begin{cases} 1 \text{ If } a_i \text{ has relation with } a_j & 1 \leq i, j \\ 0 \quad otherwise \end{cases} \quad .....4.6$$

The definition represented by equation (4.6) is twofold:

i. K_{ij} =1 if a_i has a relationship with a_j, where $a_i, a_j \in A$; otherwise,

ii. K_{ij} = 0, meaning there is no relationship between a_i and a_j.

Table 4.4: Intra-membership Relation Matrix

a_i/a_j	a_1	a_2	a_3	a_4
a_1	1/0	1/0	1/0	1/0
a_2	1/0	1/0	1/0	1/0
a_3	1/0	1/0	1/0	1/0
a_4	1/0	1/0	1/0	1/0

The intra-class membership relation matrix is used to represent the relationship within a class and its members. The matrix is captured in Table 4.4 and the intra-class membership relation matrix for Figure 4.3 is captured in Figure 4.5.

A_i/A_j	d	A()	M1()	M2()
d	1	0	0	0
A()	0	1	0	0
M1()	1	0	1	0
M2()	1	0	1	1

B_i/B_j	a	B()	M3()	M4()
a	1	0	0	0
B()	0	1	0	0
M3()	1	0	1	0
M4()	1	0	1	1

C_i/C_j	B k	C()	M5
B k	1	0	0
C()	0	1	0
M5()	0	0	1

D_i/D_j	q	D()	M6()
q	1	0	0
D()	0	1	0
M6()	1	0	1

Figure 4.5 Intra-membership Relation Matrixes for Figure 4.3

Considering OOComDN-1 for the example java program captured in Figure 4.3 the intra-membership relationship matrix which is used to represent the relationship between elements in each class given by: A_i/A_j, B_i/B_j, C_i/C_j and D_i/D_j for A, B, C, and D respectively. For instance, in A_i/A_j, all the zero values indicate that there is no relationship between members within the classes, while 1 indicates the presence of a relationship.

Definition 4.8: *[Inter- membership Relation]*

Given the ***OOComDN-1****, two classes* $A=\{a_1,a_2,\ldots,a_m\}$, $B=\{b_1,b_2,\ldots,b_n\} \in N$, $A \neq B$, *and a dependency* $"+" \in D^E$ *for* $A \rightarrow B$. *We then define the inter-membership relation matrix for dependency* ***E*** *as follows:*

$$P_{A\rightarrow B} = [p_{ij}] = \begin{cases} 1 \text{ If } a_i \text{has relation with } b_j \\ 0 \text{ otherwise} \end{cases} \quad 1 \leq i \leq m, 1 \leq j \leq n \quad \ldots\ldots\ldots\ldots\ldots\ldots 4.7$$

Table 4.5 Inter-membership Relation Matrix

a_i/b_j	a_1	a_2	b_3	a_n
b_1	1/0	1/0	1/0	1/0
b_2	1/0	1/0	1/0	1/0
b_3	1/0	1/0	1/0	1/0
b_m	1/0	1/0	1/0	1/0

Table 4.5 shows the inter-class membership relation matrix for a_i and b_j. The above definition indicates that $p_{ij} = 1$ if a_i in class A has a relation with b_j in class B, where $a_i \epsilon A$, $b_j \epsilon B$; otherwise $p_{ij}=0$ indicating there is no relationship that exists. This representation is captured in Figure 4.6. The relationships are given by A_i/D_j, B_i/D_j and A_i/C_j in the matrix. Like MRM, all the zero values indicate that there is no relationship between members of the two corresponding classes while the value 1 indicates the existence of a relationship.

A_i/D_j	d	A()	M1()	M2()
q	0	0	0	0
D()	0	0	0	0
M6()	0	0	1	0

B_i/D_j	a	B()	M3()	M4()
d	0	0	0	0
M1()	0	0	0	0
M2()	0	0	1	0

A_i/C_j	d	M1()	M2()
B k	0	0	0
C()	0	0	0
M5()	0	0	0

Figure 4.6 Inter-membership Relation Matrixes for Figure 4.3

All these matrices: CDM, MRM and IRM will be used in the chapter that follows in the computation of the SIS.

4.5 Experimental Analysis

This section reports on the results from the *controlled* experiments performed in this research. The goal that underlies the *quasi* or *controlled* experiment was to demonstrate the importance of CIA in software maintenance and the objective is twofold:

1. To demonstrate that a good and effective representation of an OO program can increase the understandability and enable the maintainer to perform modification tasks successfully. To be able to maintain and change a system efficiently and correctly, the maintainer needs to have an in-depth understanding of the systems' structure (source code). By *efficiency*, we mean the minimum time taken to carry out the change while *correctness* is the intended functionality and of the change with as few side-effects as possible.

2. To demonstrate that using a mixture of program comprehension models: *opportunistic* and *as-needed* models, a maintainer will be able to conduct maintenance tasks efficiently with less cost in terms of time, effort and number of resultant errors introduced during the change.

The first objective is aimed at evaluating the impact a good representation of an OO program on maintenance, CIA in particular. The evaluation is based on the time taken to perform the maintenance tasks and the effort expended to correctly perform the task. The second objective evaluates the impact of program comprehension on maintenance. To achieve these objectives, the *controlled* experiment was performed in two phases:

1) Phase I: In this phase, maintenance was conducted neither with the use of IR of OO program nor the use of any impact analysis model. Modifications were strictly performed directly on the source code based on the developers' experience or understanding of the program execution.
2) Phase II: In this phase, the developers performed modifications on the OO program source code after a careful study of the program and the change request supported using the program's IR that represents its structures in terms of components and dependencies.

In both cases, the essence was to measure the duration, the number of resultant errors and percentage of correctness of a modification. An overview of the experimental design is captured in Figure 4.7.

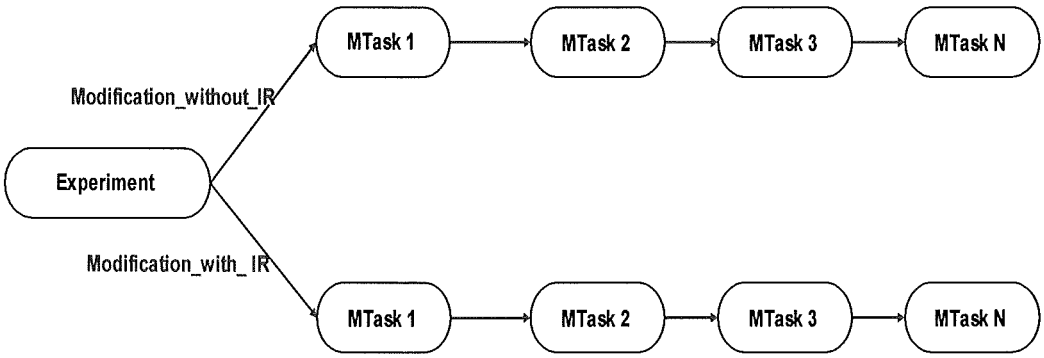


Figure 4.7: Experimental Design Structure

Figure 4.7 presents the structure of the experiment showing the phases involved: *modification-_without_IR* and *modification_with_IR* for phase I and phase II respectively. In each case, the maintenance tasks, MTask1 to N were carried out by the subjects and each phase was carried out separately. However, in all the phases the tasks ($MTask_1$ to $MTak_N$) were the same. The subjects were not pre-informed of any maintenance task or the system they were going to maintain. This was necessary to assess their existing understanding and maintenance skills.

4.5.1 Study Hypotheses

The hypotheses that were tested in the experiment are presented in this section. The aim was to assess the effectiveness of the IR during the maintenance task, i.e. did it have positive, negative or no effects on the time required to perform a modification correctly.

Thus, the null hypotheses of the experiment were as follows:

1. **Impact of TaskPhase on ChangeDuration (CD)**

   $H0_1$: *The time taken to perform maintenance task is equal for* ***modification_without_IR*** *and* ***modification_with_IR***

   $\mu_{modification\_without\_IR\ CD} = \mu_{modification\_with\_IR\ CD}$

2. **Impact of TaskPhase on Number of Errors Introduced:**

   $H0_2$: *The number of error introduced in a changed program is equal for* ***modification-_without_IR*** *and* ***modification_with_IR***

   $\mu_{modification\_without\_IR\ NoE} = \mu_{modification\_with\_IR\ NoE}$

3. **Impact of TaskPhase on Program_Correctness (PC)**

   $H0_3$: *The correctness of the program after maintenance task is the same for both* ***modification_without_IR*** *and* ***modification_with_IR***

   $\mu_{modification\_without\_IR\ PC} = \mu_{modification\_with\_IR\ PC}$

The effect on duration (CD) is designed to evaluate if using IR constitutes a waste of time or not on the part of the maintainer while the effect of correctness (PC) is to evaluate if IR contributes to program understanding or not. If PC is equal for both, then it is not useful for CIA. However, if the PC is more for *modification_with_IR* than *modification_without_IR*, then it is useful for CIA and aids comprehension of the program as well. Furthermore, for the number of errors introduced (NoE), the task would be to test if the number of errors introduced after modification is equal in both cases or not. If it is lower with the TaskPhase, *modification_with_IR,* then it is useful for CIA, otherwise it is not useful. The statistical technique used to test the hypothesis is discussed in subsequent sections. The conceptual model of the experiment is captured in Figure 4.8.

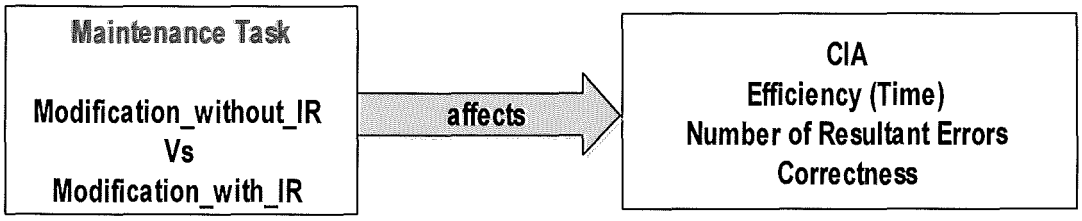


Fig. 4.8: Study Conceptual Model

4.5.2 Study Subjects and Settings

Participants of this experiment were only undergraduate students of Computer Science and Information System department, University of Venda. The study was in fulfillment of the module Software Engineering academic curriculum. The focus was on software maintenance techniques and the importance of CIA in software maintenance. As part of their semester assessments in the first semester the students were asked to develop the same small to medium-sized systems, a banking system as one of their projects. In this project, the students were given one month to develop their system. About 55 students participated but after careful consideration based on the quality of the systems built, only 45 students (9 teams) projects were considered for this analysis, with 5 students in each team, these students were in their final year (3^{rd} year).

All the students had comparable levels of education and experience in software development and Java programming. Though there were differences in skills, each team was blended with the required skills needed. To help them in carrying out the maintenance, the subjects had a week of theoretical knowledge on software maintenance, the basic knowledge needed for CIA using IR of OO program and so on.

4.5.3 Material and Data Collection

The students were asked to develop the system using java programming language and following strictly the waterfall model of software development. Each team was asked to submit the deliverable of each phase ranging from specification documents to test cases. The teams were instructed as follows:

- Use reusable components where necessary but without the java standard library classes and develop non-graphical user interface (GUI) system. This is because there are difficulties in comprehending GUI implementation [129].
- Arrange their work into packages where applicable, and
- Provide complete documentation of their work.

At the end of the project, the systems were acceptance-tested by the lecturer responsible during submission. For each team, the number of classes in each system collected was computed. The systems developed by team A, D, F, H, and I had five classes each while the systems of team B, C, E, and G had six classes each.

4.5.4 Maintenance Tasks

Each team was asked to swap their system with their neighbor in order to maintain the other team's system. The subjects were asked to perform four maintenance tasks during the course of the experiment:

i. $MTask_1$ - *one class change*

ii. $MTask_2$ - *one class change*

iii. $MTask_3$ - *two methods change*

iv. $MTask_4$ - *one field change*

The changes were based on the different change types (*chtype*) applicable for OO programs.

The maintenance tasks for the phases were as follows:

- Phase I: In this phase, the teams were asked to make 5 changes to the systems as stated in the MTasks. They were given about 10min to study the program before implementing the changes. These changes were all made simultaneously without the use of IR of the OO program or existing CIA model.

- Phase II: In this phase, the same changes were performed, but this time they were given the IR of the program to be used for the maintenance task. Modification was performed using the program comprehension model shown in Figure 3.5 of chapter 3.

Both phases of the experiments were conducted in the computer laboratory under the supervision of the lecturer responsible for the course.

4.5.5 Variables and Statistical Technique

4.5.5.1 Variables

In the experiment, the variables that were of importance at each phase of the maintenance task are the change duration, program correctness, the number of errors the change introduced and the task phase:

1. ***Change Duration:*** With change duration, before the modification task begins, we wrote down the start-time and when they completed the task, the stopping-time was taken. We then computed the total time taken for the maintenance tasks in minutes called change duration, CD. This was done manually by the lecturer responsible and for this to be effective and meaningful, only time spent on this task was considered.

2. ***No of Errors:*** We computed the numbers of errors (NoE) introduced by the modification task after the changes were made by recompiling the program. In this case, numbers of errors were computed based on the number of lines affected as indicated on the development IDE used. This was performed by the supervisor and the team members.

3. ***Program Correctness:*** The program correctness, PC was computed based on the correct implementation of the task based on the difference between the expected output and the actual output produced by each program after all errors were corrected. This was purely done by the supervisor. Each team was graded between 0-100% based on the outcome of the tasks and the program execution.

4. ***TaskPhase:*** This variable describes whether the students carried out the change tasks using the *Modification_without_IR* ($MTask_1$, $MTask_2$, ..., $MTask_n$) or *Modification_with_IR* ($MTask_1$, $MTask_2$,... $MTask_n$). Depending on the programming skills of the subjects, each team's program was first assessed for actual amount and complexity of classes that would be impacted by each change and the approximate time required to carry out the tasks. This was necessary in order to quantify the degree of difficulty of the change tasks. However, the results obtained indicated that the approach was adequately appropriate. (see Section 4.6)

4.5.5.2 Statistical Technique and Specifications

In this study, we used the paired-sample T-test called the dependent T- test statistical technique to test the hypotheses stated in Section 4.5.1. A paired-sample is used to analyze paired scores to determine if a difference exists between them. It is used in comparing measurements from the same participants by using two different measurement approaches. The explanation of the variables used is given in Section 4.5.1. The statistical software package SPSS was employed in this case to test the hypotheses. The motivation for the choice of the dependent T-test is that it offers a flexible approach for measuring the effectiveness of two different techniques using the same participants. *Modification_without_IR* and Modification_*with_IR* are the measurement techniques that were used. This was necessary to evaluate the effectiveness of the OO program's IR. In addition, the dependent T-test is used to determine if two means are different from each other when the two samples that the means are based on were taken from same participant.

All the variables specified were normally distributed. We used the Shapiro-Wilk Test since it is appropriate for small sample sizes, say less than 50 (< 50) [49]. There was no transformation performed on the variables since they have no potential negative effect. The model specification is captured in Table 4.6. In the event that the underlying assumptions of the models are not violated, the related null hypothesis will be rejected if the presence of a significant model term corresponds to $p \leq 0.05$.

Table 4.6: Statistical Technique Specification

Variable	Distribution	Model Term	Use of Model Term
Duration	normal	TaskPhase	Test $H0_1$
No. of Errors	normal	TaskPhase	Test $H0_2$
Program Correctness	normal	TaskPhase	Test $H0_3$

In Table 4.6, a more detailed explanation with respect to the term used to test the hypotheses is that the TaskPhase variable was used to model the main effect of the *Modification_without_IR* tasks versus *Modification_with_IR* tasks on duration, the number of faults introduced and correctness of the program (to test hypotheses $H0_1$, $H0_2$ and $H0_3$).

4.6 Results

In this section, we present the results from the experiments that were performed.

4.6.1 Descriptive Statistics

The descriptive statistics of the experiment are captured in Table 4.7 and Table 4.8 for PHASE I and PHASE II respectively and the variables, PC, NoE and CD are reported as well. In both tables, PC contains the grade given to each Team in percentage (%) based on the correctness of their modification task (i.e. a program is correct if all modification tasks are executed correctly), NoE column shows the number of errors the modification introduced after the changes were made, while the duration is captured by CD. In addition, the mean, μ and the standard deviation statistics are reported.

Table 4.7: Descriptive Statistics for PHASE I

Variable	Min	Max	Mean	Std. Deviation
PC(%)	29.00	52.00	42.6667	8.77496
NoE	16.00	26.00	21.4444	3.53946
CD(min)	38.00	56.00	45.5556	7.01981

Table 4.8: Descriptive Statistics for PHASE II

Variable	Min	Max	Mean	Std. Deviation
PCI(%)	51	89	70.44	12.768
NoEII	0	9	5.22	3.346
CDII (min)	30	39	33.44	2.877

The main results based on the task phases (modification_without_IR and modification_with_IR) for MTask1 – Mtask4 are visualized in Figure. 4.9 and Figure 4.10 using a line graph representation.

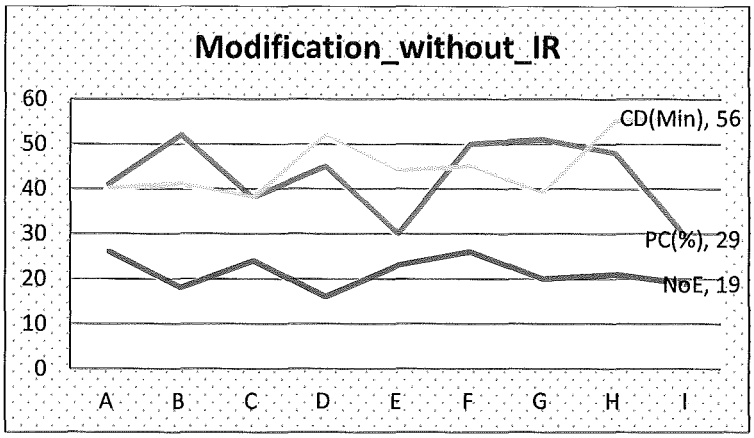


Figure 4.9: Effect of TaskPhase on CD, PC and NoE

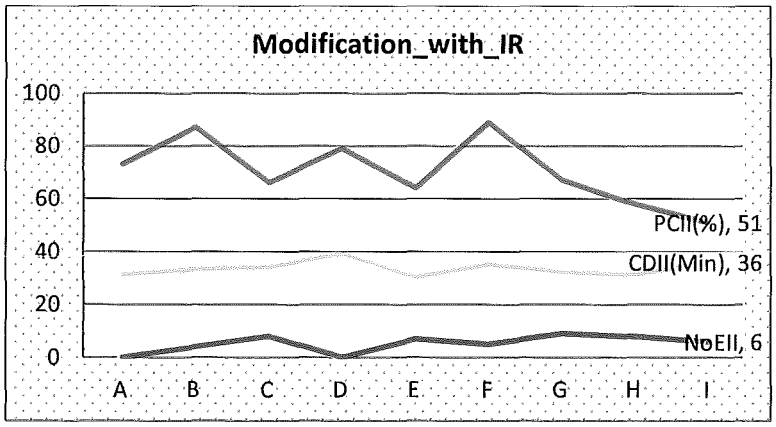


Figure 4.10: Effects of TaskPhase on CDII, PCII and NoEII

In both figures, the CD, % PC and NoE are shown on the Y-axis, while the project groups are shown on the X-axis. As we can see, there are clear indications that TaskPhase affects CD, NoE and PC. For example, it is clear that a smaller amount of time was utilized to make changes on a program when IR was used (see Figure 4.10). In the same way, PC was better when IR was used in the maintenance task first. The same result is applicable to NoE introduced in each phase. However, for practical importance, it is necessary to see if the differences are significant by testing the above specified hypotheses.

4.6.2 Hypothesis Tests

Firstly, the paired-sample T-test validity test was conducted and the results of the normality test are captured in Table 4.9. Paired-sample T-test is valid if a normal distribution is assumed. In the case of Shapiro-Wilk Test, pair values are normal if $p \geq 0.05$. The results show that the pair values were normally distributed.

Table 4.9: Test of Normality

Paired Value	Kolmogorov-Smirnov[a]	Shapiro-Wilk
	Sig.	Sig.
DC	.200	.192
DCII	.200	.814
PC	.200	.689
PCII	.200	.155
NoE	.200	.130
NoEII	.200	.614

Accordingly, Figure 4.11a, Figure 4.11b and Figure 4.11c are the graphical representations of the normality on a normal Q-Q Plot. As shown on the graph, the data points are all seen being close to the diagonal line proving that the data are normally distributed. This satisfies the requirements for a paired-sample T-test.

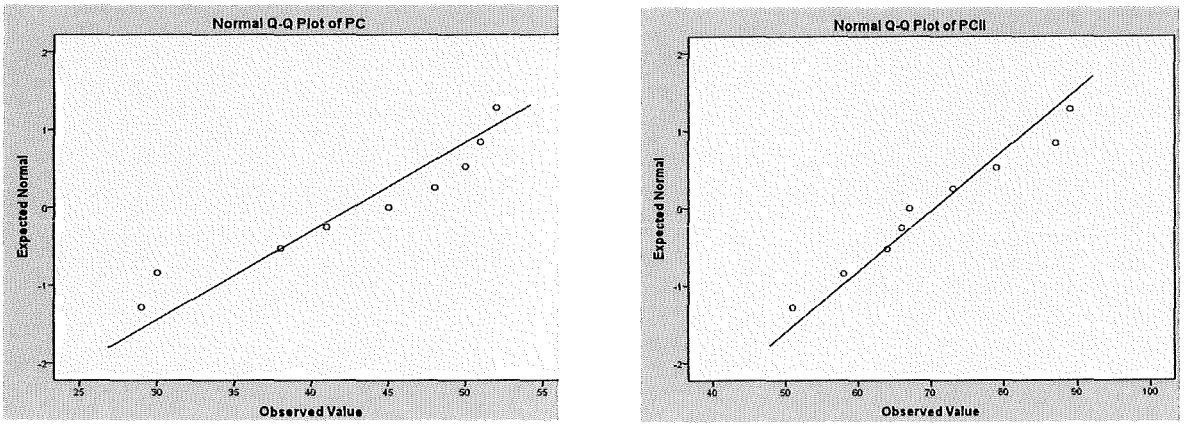


Figure 4.11a: Normal Q-Q Plot for PC vs PCII

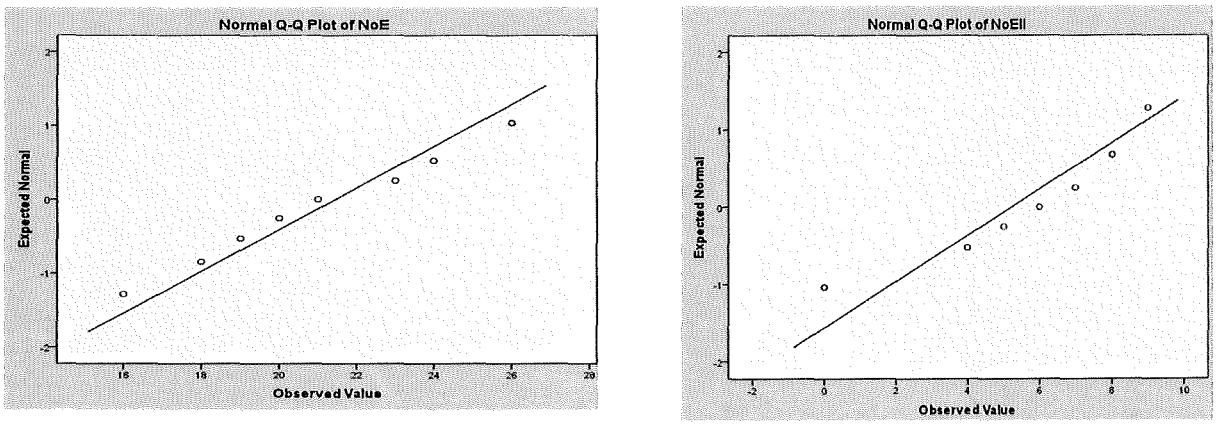


Figure 4.11b: Normal Q-Q Plot for NoEvsNoEII

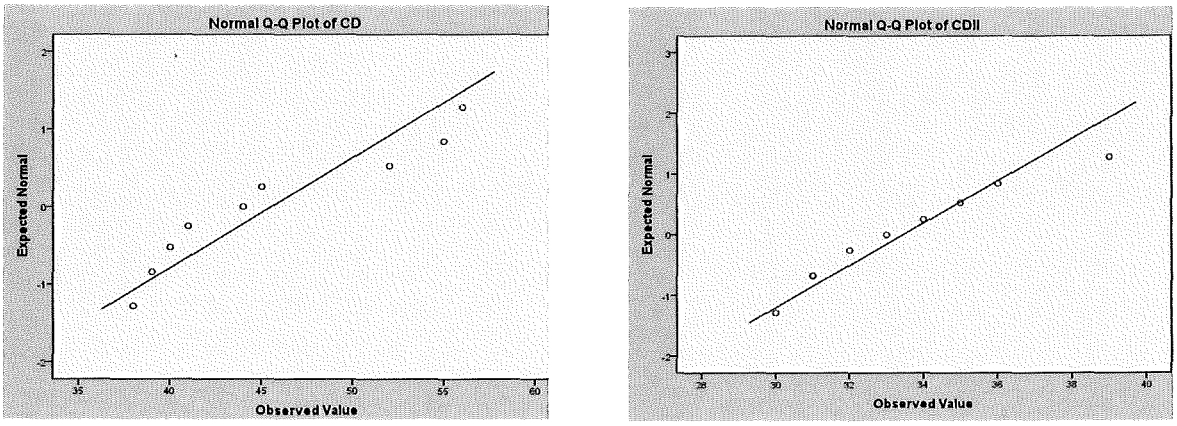


Figure 4.11c: Normal Q-Q Plot for CD vs CDII

Next are the results of the hypotheses regarding CD, PC and NoE introduced for the maintenance tasks ($MTask_1$-$MTask_4$). For both TaskPhases, the results are captured in Table 4.10. The results indicate that TaskPhase does have a significant effect on the program correctness, change duration and number of errors introduced. In this case, the hypothesis was tested at a significance of $p \leq 0.05$.

Table 4.10: Dependent T-test Results Regarding Change CD, PC, and NoE (MTask1-MTask4)

Paired variable	T	DF	P-value Sig(2-tailed)
CD - CDII	-8.541	8	0.000
NoE - NoEII	10.509	8	0.000
PC - PCII	5.646	8	0.000

The results of the hypothesis tests are summarize as follows:

For the impact of TaskPhase on CD:

$H0_1$: *The time taken to perform the maintenance task is equal for modification_without_IR and modification_with_IR.*

Reject: We reject the $H0_1$ since p-value $\approx 0.00 \leq 0.05$.

For the impact of TaskPhase on NoE:

$H0_2$: *The number of errors introduced in a changed program is equal for modification_without_IR and modification_with_IR.*

Reject: We also reject the $H0_2$ since p-value $\approx 0.00 \leq 0.05$.

For the impact of TaskPhase on PC:

$H0_3$: *The correctness of the program after maintenance task is the same for both modification_without_IR and modification_with_IR.*

Reject: We reject the $H0_3$ since p-value $\approx 0.00 \leq 0.05$.

In conclusion, at the $\alpha = 0.05$ level of significance, there exists enough evidence that there is a huge difference in the mean change duration, program correctness and the number of errors introduced after changes were made on both phases of the maintenance. These results thus prove that the IR for OO program's representation is effective and useful in facilitating CIA.

4.7 Results Discussion

The results of the experiment seem very interesting in terms of duration, program correctness and the number of errors introduced after change were effected for phase II. As captured in Figure 4.8, it is clear that the time taken by the subjects to perform the maintenance task in phase II (36 min maximum) were significantly smaller than the modification duration for phase I (56 min maximum) as shown in Figure 4.7. Accordingly, the correctness of the maintenance task (correct solutions) was significantly higher for phase II (56% minimum) than for the phase I (51% minimum). Moreover, the number of errors introduced after the changes were made was significantly higher for phase II (6 maximum) when the TaskPhase was *modification_with_IR* as opposed to *modification_without_IR* (19 minimum).

The results suggest the effectiveness of the IR for CIA. Using the IR of OO program will actually reduce the time needed to make changes, and the number of errors that will be introduced after the change and increase the correctness of the solution. However, the interpretation of these results requires care. This is because, though we took good time to blend each team with skillful and experienced subjects, the experiment actually did not take care of such experiences and skills in terms of the team. The level of skill and experience of each team differed and could affect the maintenance task in terms of efficiency and comprehension. Factors that could also affect the results are the system's structural properties such as coupling, cohesion and inheritance. Naturally, a good design involves having low coupling and high cohesion in a system for maintenance to be effective. Unfortunately, the reverse of these design properties (high coupling and low cohesion) have been known to have natural effects on change propagation across systems. Consequently, much time could be spent by each team on comprehending and performing changes correctly. In addition, while some errors still remained in most of the team's programs after changes were made this could be as a result of either undiscovered indirect impacts resulting from the system's structural properties or the programming experience of the subjects.
Another possible reason for the effects of the TaskPhase on CD, PC and NoE is the program comprehension strategy employed during the maintenance tasks. Based on the program comprehension model discussed in chapter 3,and taking Littman et al. [74] breadth of comprehension strategies: *systematic* and *as-needed*. The integration of the as-needed strategy

into the model is commendable. Though several studies have shown that the application of the systematic strategy is more likely to produce correct solutions in terms of knowledge gain and understanding before modifications [129][130], the use of the as-needed strategy alongside the OO program's IR in phase II helps the subjects to minimize the amount of code to be understood before changing the program. This consequently, results in reduced modification time, high correction rate and low error rate.

In phase I the subjects tended to spend time using their knowledge of OO programming to first understand most aspects of the system before changes were made. Consequently, much time was wasted in performing changes to the system. Though subjects had a good overview of the system in order to make the changes in phase I, the subjects in phase II had a better overview of the system through IR and only had a minimum number of codes to understand to make modifications successfully.

Based on the theories of program comprehension in the opportunistic model, we can characterize phase I as representing a top-down and systematic approach to program comprehension. Accordingly, phase II represents a bottom-up and as-needed approach to program comprehension. Thus, for practical application of the IR during CIA process, the experiment shows there is a significant difference in the time spent to correctly maintain an OO program when IR of the system is used and this also impacts on the program correctness and the number of faults introduced. This is due to the fact that the top-down approach is typically applicable when the system to be maintained is familiar to the maintainer [130]. In summary, the results suggest that employing *the opportunistic-as-needed comprehension model* would help programmers (experienced or novices) having little prior knowledge of the system to be maintained to correctly implement modifications in a system.

4.8 Threats to Validity

Experiments are always associated with potential risks that can affect the validity of results. In this section, we discuss the most important possible threats to the validity of the quasi experiment and what has been done to reduce them.

4.8.1 Internal validity

Internal validity threats are effects that can affect the independent variable (TaskPhase) with respect to causality, without the knowledge of the researcher's in an experiment [16]. They pose a threat to the conclusion about a possible causal relationship between treatment and outcome. In this thesis, the experiment was performed in two phases and in the same location and setting. Thus, lack of randomization of the TaskPhase assignment could result in skill differences between

the participating teams, which in turn would render the results biased. However, in order to address this potential threat, we assigned each subject to a team based on their previous performances to ensure that each team was balanced. In addition, since the same participants were involved in both phases, the dependent t-test proved most suitable for testing the stated hypotheses.

4.8.2 Construct validity

Construct validity deals with the degree to which conclusions are justified from the perspective of the observed participants, study settings, and dependent and independent variables. The validity threats are as follows:

A. Measuring Program Correctness, Change duration and Number of faults

In the experiment, three simple measures were used as dependent variables: PC, CD and NoE. The variable PC, a measure of program correctness, was a mark given that shows whether the subjects obtained a correct solution after change tasks MTask1 – 4 were carried out. To show the quality of the marks given, an independent expert was consulted. The programs were thoroughly tested and the program code was also inspected. We believe this assisted in ensuring that the program correctness measure was appropriate. The CD which measured the time spent to perform maintenance tasks correctly for modification tasks MTask1 – 4. Though time was measured as a difference between finish time and start time, we believe it might be affected by factors such as calling the attention of the supervisor and so on, during the experiment. However, we took every step to reduce this threat. Also, NoE is a count of the number of faults found on the IDE after implementing the changes for modification tasks MTask1 – 4. During compilation, necessary steps were taken to count the actual faults that originated. In addition, though PC, CD and NoE are important pointers of program maintainability that reflect maintenance cost, however, several other maintainability dimensions were not covered such as faults severity, the design quality of the program and so on. To eliminate these threats, we only selected quality programs for the experiment.

B. Task Phase

The division of the experiment into phases; *modification_without_IR* and *modification_with_IR* could be another important threat to the construct validity in the experiment. In this case, the trend was to determine whether the variable TaskPhase has satisfactory construct validity. In the context of the experiment, to check the construct validity we quantified beforehand the difficulty of modification tasks in terms of amount of class each program had and their complexity and the time needed to implement the changes.

4.8.3 External validity

The threats to external validity concern conditions that limit generalization of the results obtained in the experiment [16][128]. Such threats are mainly from the participants, the settings and the nature of the system maintained.

A. Application and Tasks

The systems used for the experiment were very small in size, maximum of two packages, 6 classes which were not up to thousands LOC. Thus they were small-sized applications compared with industrial OO program systems. In addition, the modification tasks were relatively simple, small in size and time. Program characterized in this manner poses limitation serious to controlled experiments and is dependent on the research question being asked as well as to the extent to which the results are supported by theory [16][128]. In the experiment, we showed a clear impact of TaskPhase, notwithstanding the small size of the applications and modification tasks. Its generalization to larger applications and tasks can be made with the support of existing program comprehension research theories. We provide such support in Section 4.6.

In addition, it is possible that the task phases, comprehension strategies and their effects on project team's performance would be different for larger systems and complex maintenance tasks since larger systems will often require larger cognitive complexity. Also, if the experiment had lasted longer (i.e. time to become familiar with the system), the results may have been different.

B. Subject sample

All the participants used in the experiment were only undergraduate students of computer science and thus fell in the class of "novices" or as "advanced beginners" as stipulated by [131]. Similar results might also be obtained by subjects having a similar background. Due to the small sample size of about 45 in nine teams involved, caution is needed when interpreting the results. Also participants varied because of their individual programming skills and experience. However, due to the blending of the teams with skillful and experienced subjects, it is believed the presence of differences had no significant impact on the results obtained.

4.9 Chapter Summary

In this Chapter, we have presented the different approaches aimed at representing OO software systems in terms of change diffusion, fault diffusion and dependency computation. The first approach is to proffer an understanding of the source code by visualizing its structures, components and dependencies. We used the concept of complex software network called OOComDN to discover the topological structures of the software which in turn will assist in identifying true impact sets during CIA. Secondly, the chapter introduced an approach to assess

the risk each connected components has on others in terms of fault propagation. Lastly, a novel approach of representing the relationships and dependencies among classes and its members using adjacency matrix was introduced. This will assist a maintainer to discover the starting impact sets of a change at a high-level of abstraction. In addition, the first approach was also evaluated and the results obtained indicate that the IR of the OO program can be helpful in facilitating change impact analysis in terms of program comprehension, duration and correctness.

CHAPTER 5

Impact Prediction Technique

5.1 Introduction

This chapter deals with the practical aspects of OO program impact predictions. It begins with an overview of the impact model and then advances to the impact computation strategy where ripple-effect analysis is performed. The basic concepts of the impact analysis are the mechanisms for computing *impact sets* and their *expressions* ranging from SIS to EIS. Moreover, the methods for computing the precisions of the impact techniques in order to determine its effectiveness are discussed.

5.1.1 Overview

During software maintenance, having a good understanding of the structure of the program to be changed will make the task easy and effective. In this thesis, we consider software components: fields, methods and classes, as pieces of codes that may undergo changes. These components are typically entities of analysis at a granular level. During the course of modification on a software component, the task of CIA therefore, is to identify additional components that will be affected by the change. The resultant set which is comprised of the starting and affected components is what is referred to as *impact set* (IS). The conceptual idea is formally expressed as follows:

Definition 5.1:*[Impact of a Change]*

Let ***P*** *be a java program represented by a set of components or entities* $e_p = \{e_1,e_2,..,e_n\}$. *Let* $chtype = \{ chtype_1, chtype_2,..., chtype_n\}$ *be the set of source code change types that can be performed on the components of* ***P*** *and* $d^{type} \in D^{type}$ *the dependencies between the components in* ***P*** *such that for a given change, the impact set is defined as:*

$$P_{impact}\{ chtype_i, e_i, d^{type}_i \} \rightarrow \{ e_1,.....e_i,e_k,.... \}$$

Such that after the change is implemented in **P**, the system will still operate properly and the affected components $\{ e_1,.....e_i,e_k,...\}$ changed as required.

Where:

- $\mathbf{d^{type}_i}$ represent dependencies types that links the various component of the program, $\mathbf{e_p}$
- $\mathbf{chtype_i}$ depend on the OO software system's components;
- $\mathbf{P_{impact}}$ is the programs' impact function; and
- $\{ e_1,.....e_i,e_k,.... \}$,the impact sets which are comprised of components that have both direct and indirect relationships through $\mathbf{d^{type}_i}$ with $\mathbf{e_i}$

To extract d^{type}_i and predict accurately which components are candidates of $\{ e_1, \ldots\ldots e_i, e_k, \ldots \}$, OOComDN is used. This is discussed in the sections that follow.

5.2 Change Impact Techniques

Due to the complex dependencies that exist in OO programs, it is essential that the impact of a change is kept as low or local as possible so that it does not propagate or bring new faults into other parts of the program when it is implemented. This will ensure the consistency of the modified program. However, to achieve this requires an effective approach to precisely predict which components will be affected by the change in order to take prompt decisions on whether to commit the change or not at the appropriate time.

This section, in addition to the cognitive model and dependency analysis/extraction approach discussed in previous chapters, will deal with the impact prediction technique that will predict the impact set of a given change. The approach assumes that the key to effective CIA on an OO program lies on:

1. Developing a good mental model of the software system,
2. Effective representation and extraction of various dependencies of the software system,
3. Knowing the various source code change types, dependencies and the associated impact range of each change type.

The task therefore, is to describe how various dependencies and categories of code changes affect the impact of a change from one component to others. With the impact function, $P_{impact,}$ it is worth knowing that when changes are considered, components that are truly affected can be identified through the:

i. Types of *source code change* involved,
ii. Kind of *dependencies* that exist between components,
iii. Impact diffusion range of each change type and dependencies.

These constitute what is termed *change impact technique* in the context of this research. The details involved are discussed in subsequent sections.

5.2.1 Effect of Code Change Types

Looking critically at OOComDN-1 presented in Figure 4.3, in the realm of OO programs, *faults* can only propagate from one component to others depending on the number of components that are linked directly or indirectly. However, for a *change*, this is not always true. The reason is that different changes have their own behavior and affect other parts of the program differently. One determinant factor for this is the existence of strong OO program features such as encapsulation in the program. For example, modifying a *field type* will affect all the classes that reference it, but

adding a new field would impact no existing classes. Accordingly, changing the *access specifier* of a field from *private* to *public* will have no impact on other fields, methods or related classes that uses its, but changing the *access specifier* from *public to private* will affect any related classes and subclasses referencing them, though members in the changed class will not be affected. Thus, the ability to recognize different changes and be able to relate them to the various categories of source code change type of an OO program is a big step towards effective CIA. When making changes that involve different *packages, classes, methods* and *fields*, it is of the essence to define the total changes that are required to be implemented to satisfy the change proposal. This is called change sets (**chset**) and is defined as follows:

Definition 5.2: ***[****Change Set (****chset****)]***

*Change Set (****chset****) is the set of all source code change type that can be implemented on an OO program for a particular change or changes.*

Accordingly, the **chset** considered in this thesis are Packages = 3, Classes = 16, member methods = 15 and member fields = 10. They are all captured in Table 3.1, 3.2, and 3.3 of Section 3.3 of chapter 3 respectively. The computation is as follows:

$$\text{chset} = \sum_{i=1}^{k}(\text{chset}_F + \text{chset}_M + \text{chset}_C + \text{chset}_{Pk}) \quad \text{.....5.1}$$

Where:

chset$_{Pk}$ = {APk, DPk, MPkN}

chset$_C$ = {CA, CD, ICA, DCA, AAbC, DAbC, CCN, APC, DPC, CID, AFC, DFC, ASC, DSC}

chset$_M$ = {AM, DM, IMA, DMA, CSM, AAbM, DAbM, MRT, MNPM, MPM, MNM, AFM, DFM, ASM, DSM}, and

chset$_F$ = {AF, DF, IFA, DFA, MFT, MFN, AFF, DFF, ASF, DSF}

5.2.2 Effect of Dependencies Types

The type of link that exists between the changed component and other components related to it either directly or indirectly is another factor which *change impact techniques* depend on. This is necessary because the existence of dependencies that link components to one another sometimes deludes engineers with the thought that a change on one component will inevitably affect others. However, in an OO program, it is not always true in the practical sense. For instance, let us consider a source code change of the type "*decreases the accessibility of a method*". This type of change involves changing a method from either *public* to *protected* or *private*. If *protected* is considered, the impact range will consist of all components that call this method except for classes which are inherited from the changed class. In this case, the inheritance dependency is immune from the impact of such change.

However, for effective prediction of *impact sets* during CIA, the identification of source code change type goes in simultaneously with the dependencies that exist because dependency type alone cannot determine the impact of all changes. In this thesis, the four identified types of dependencies will be considered. The process involves in computing *impact sets* for CIA is discussed in the section that follows.

5.3 Change Impact Analysis Process

This section discusses the processes involved in predicting the impact of a change. It starts with the **SIS** and then progresses to **EIS** as well as **AIS**.

5.3.1 Starting Impact Set

In this thesis, **SIS** is used to improve the precision of the CIA method. It is based on the assumption that the precision of **EIS** is highly dependent on the accurate estimation of **SIS.** Thus, the technique adopted in the determination of the **SIS** in this thesis is unique and is apt for the targeted audience. Based on the OOComDN-1, **SIS** is derived from the dependency matrixes discussed in Section 4.4 of chapter 4: class dependency matrix (CDM), $[m_{ij}]$ as well as the intra and inter-membership relationship matrices, $[k_{ij}]$ and $[p_{ij}]$ respectively. Basically, **SIS** is purely dependent on the operation on adjacency matrix. However, before defining what **SIS** is, let us delve into define the *impact values* of the matrixes.

Definition 5.3: *[Impact set Values (IV)]*

Let m_{ij} *be class dependency matrix CDM,* $c_i = \{c_1, c_2, \ldots c_m\}$*, two classes A, B, where A has set of members* a_i*,* $A=\{a_1, a_2, \ldots, a_m\}$*, B with set of members* b_i*,* $B=\{b_1, b_2, \ldots, b_m\}$*,* k_{ij}*, and* p_{ij}*. Therefore, the impacted set values from the matrixes are defined as follows:*

$$IV(a_i) = \sum_{j=1\ j\neq i}^{m} k_{ij} + p_{ij} + m_{ij} \quad \ldots\ldots 5.2$$

Where **i** and **j** represents the column and row values for $\mathbf{k_{ij}} \in \mathbf{K_A}$, $\mathbf{p_{ij}} \in \mathbf{P_{A\rightarrow B}}$ and $\mathbf{m_{ij}} \in \mathbf{CDM_{A\rightarrow B}}$ respectively, and $\mathbf{IV(a_i)}$ indicates:

i. The number of class members that relate to member $\mathbf{a_i}$ within a class, $\mathbf{k_{ij}}$
ii. The number of class members in $b_i \in \mathbf{B}$ that relates to member $a_i \in \mathbf{A}$, $\mathbf{p_{ij}}$, and
iii. The number of classes that relates to either **A** or **B**, $\mathbf{m_{ij}}$ indicating the number of classes that are related to class, $\mathbf{C_i}$.

For example, as shown in the intra-membership relationship matrix for class **A**, represented as $\mathbf{A_i/A_j}$ in Figure 4.5 of chapter 4, for $\mathbf{a_1}=d$, there are two members that relate to *d*, denoting that a change in *d* will require them to be changed as well. Thus, $IV(a_1) = 3$ which are d, M1 () and M2

(). Accordingly, $IV(a_3)=1$, which is M2() only. With the representation, it can be seen that a change to a component affects itself as well as others that relate to the changed entity.

In the same vein, we define the total impact sets value (IV_{total}) for m_{ij}, k_{ij} *and* p_{ij} as the total set of components within the class that need to be changed when a change is considered on a particular component. This is defined as follows:

$$IV_{total} = \sum_{i=1}^{m} IV(a_i) \quad \text{.....5.3}$$

Where $IV \in K, P, CDM,$ and $a_i \in A$. Considering the above example, if $IC(a_1)$ and $IC(a_3)$ are the only change to be performed on A, therefore,

$$IV_{total} = IV(a_1) + IV(a_3) = 3+1 = 4$$

Where "+" operator represents a union. This denotes that the change of components a_1 and a_3 will require the change of four (4) components or members of the class. In order to represent this information, the following definition is given.

Definition 5.4: *[Component change vector,(CV)]*

Given the matrices, m_{ij}, k_{ij} *and* p_{ij} *and a given class,* $A=\{ a_1, a_2, ..., a_k \}$, *we define component change vector, CV as* $< cv_1, cv_2, ..., cv_m >$, *where* cv_i *represents the change of element* a_i, $k= 1,2,..., m$ *and* $cv_i = 1$ *if the value of* a_i *requires modification, otherwise* $cv_i = 0$.

Still on the above example, the CV of the changes to $IV(a_1)$ and $IV(a_3)$ will be computed as follows:

$$\mathbf{CV} = \{CV_1\} \cup \{CV_3\} \quad \text{.....5.4}$$
$$= \{<1, 0, 1,1>, <0,0,1>\}$$

Definition 5.5: *[Starting Impact Sets (SIS)]*

Let **P** *be an OO program and* $\mathbf{G < (N, D^E), D^{type}>}$ *represent the OOComDN-1 for* **P** *and* m_{ij}, k_{ij} *and* p_{ij} *be the dependency and relationship matrices for* ***OOComDN-1*** *respectively. If a component for instance* **e**, *and the corresponding node* $\mathbf{n \in N}$ *in OOComDN-1 is changed with* **chtype**, *where* $\mathbf{chtype \in chset}$, *SIS is therefore defined as:*

$$SIS_{\{n\}} = \{ <n, n_i> \in D^E\}$$

If m_{ij} is the class dependency matrix used at the class level, k_{ij} and p_{ij} represents the *intra* and *inter-membership relation matrix* respectively, based on equation (5.4), we can modify the **SIS** definition to:

$$SIS_n = \bigcup_{i=1}^{m} (CV_i) \quad \text{.....5.5}$$

Thus, the SIS is {<1,0,1,1>, <0,1,0>}, which include $\{d, M1(), M2(), M6()\}$. In the section that follows, we will define **EIS** as well as the factors on which it depends.

5.4 Estimated Impact Set

In the computation of SIS, we showed that it purely depends on the operation of the adjacency matrix on the changed component without reference to the *change* and dependency types between the changed component or their associated impacts. That approach was chosen in order to be effective in correctly identifying the **SIS** candidate components. However, this approach has the tendency to produce *impact sets* which are extremely large for practical use and including components not truly affected by the change. Hence, the task in this section is to refine the generated impact sets by taking into account the source code change types, dependencies behaviors and the impact diffusion range of each type of component change. **EIS** constitutes the set of components that could potentially be impacted by a change of one or more components.

5.4.1 Impact Diffusion Range Based of Change Types

Impact diffusion range (***IDff***) is the name given to the ripple-effect of a software component when changes are made on some components. Unlike structured programming, OO program approach involves message passing where some operations are demanded on the object. Thus, when these components are changed, it could potentially affect other classes [24]. The way these components are potentially affected is highly dependent on the encapsulation strength that exists. Its existence in a program influences the number of components that could be affected by a change. Thus, an excellent use of encapsulation is critical to software maintenance. In this section we present the impact associated with every change category discussed in chapter 3 and propose look-up tables (LT) to be used during CIA as reference tables.

5.4.1.1 Field Change Impact Diffusion Range

In this subsection, we describe the different kind of changes, *chtype* and their ***IDff*** that can be implemented on a class member field. As presented below, the abbreviation on the left hand side represents the field change type while on the right is a description of the ***IDff*** associated with such change type.

Change	**Impact Range Description**
AF	Adding new *data members* to a class has no impact on the existing class since it does not have any classes to use it. However, the new data member has to be known by the class descendants if it is public.
DF	Deleting *data members* will affect all other members, classes and subclasses that use, invoke or inherit them since they will no longer be available. However: • For *public* field members, only members of the changed class, descendants and other classes will be impacted by a change.

- For *private* field members, only members of the changed class will be impacted.

IFA Changing *data members* from *Private* to *Public* will have no visible impact on any member or class other than exposing the object state.

DFA Changing *data members* from *Public* to *Private* impacts any related classes and subclasses referencing them. Members in the changed class will not be impacted by this change.

MFT Changing the *type* of data members will affect the members that reference them. However:

- For *public* data member, members from the changed class, related classes and subclasses will be impacted.
- If it is private, it will affect the data members or methods from the changed class only.

MFN Changing the *name* of data members will impact all members, classes and subclasses related to them, if it is *public*.

AFF Adding the *Final Modifier* to data members denotes the field cannot have its value modified and will have impacts on all members, classes and subclasses that are related to them.

DFF Deleting the *Final Modifier* from data members denotes the field value can be modified and will not impact all members, classes and subclasses that are related to it.

ASF Adding the *Static Modifier* to data members denotes the field cannot be accessed through object and will affect all members, classes and subclasses that are related to it.

DSF Deleting the *Static Modifier* from data members denotes the field can be accessed through object and will affect all members, classes and subclasses that are related to it.

5.4.1.2 Method Change Impact Diffusion Range

The following constitutes the kind of changes, *chtype* and their ***IDff*** that can be implemented on a class member method. Again, the abbreviation on the left hand side represents the method change type while on the right is a description of the IR associated with such change type.

Change	**Impact Range Description**
AM	Adding *methods* will have no effect on existing methods or classes. However, descendants of the class need to know the new method.
DM	Deleting existing *methods* makes them not available to methods or classes that invoke or inherit them, thus, a change will affect them all. However:

- For *public* methods, the class members in the changed class, others that uses and invoke, and the descendants of the class will all be impacted.
- But if is *private*, only the class members in the change class will be impacted.

IMA Changing *methods* from *Private* to *Public*, no other methods or classes that invoke or inherit them will be affected except where the state of the object is exposed.

DMA Changing *methods* from *Public* to *Private*, will impact all classes and subclasses that reference them since the methods will no longer be visible to them.

CSM Changing a *statement* within a method body will affect the class children and other member methods that invoke it.

MRM Modifying methods *Return Type* will impact on all the classes and class descendants that inherit or invoke them.

MNPM Modifying the *Name of Parameters* will only have internal and external impact on statements inside the method body and outside entities that reference it respectively.

MPM Modifying methods *Parameters* without methods name will affect the subclasses and other methods invoking or invoked by the method.

MNM Modifying methods *name* impacts any class member relating to the methods. However:

- For *public* methods, class members from the changed class, classes that call and inherit them will be affected by the change.
- But for *private* methods, only class members from the changed class will be impacted by the change.

AAbM Adding the *modifier "abstract"* to existing methods will impact the classes and subclasses that inherit, invoke and uses them as well as class members that depend on them.

DAbM Deleting the *modifier "abstract"* from existing methods impacts every class and subclass that inherits and invokes them as well as the class members relating to them.

AFM Adding the *Final Modifier* to methods means the methods cannot be overridden by the subclasses of the changed class. Hence, the change will affect them.

DFM Deleting the *Final Modifier* from methods means the method will become visible to all subclasses from the changed class. Hence, the change will not affect them in any way.

ASM Adding the *Static Modifier* to methods will affect classes and subclasses related and calling them.

DSM Deleting the *Static Modifier* from methods will affect no classes or subclasses related and calling them.

5.4.1.3 Class Change Impact Diffusion Range

Just like the field and method *chtype*, the kind of changes and their ***IDff*** that can be implemented on a class are presented in this section. The abbreviation on the left hand side represents the class change type while on the right is a description of the impact diffusion range associated with such change type.

Change	Impact Range Description
CA	Adding *new classes* will have no impact on any other classes, since the existing classes are yet to use them.
CD	Deleting *existing classes* means that the classes and all their members will no longer be available to other related classes. Thus, all the classes related to them will be impacted by the change.
ICA	Modifying the *access specifier* of classes from *private* to *public* means the classes will become visible to all classes, thus they will have no impact on their members, other classes and subclasses that relate to them.
DCA	Modifying the *access modifier* of classes from *public* to *private* means the classes will no longer be visible to all the classes and subclasses that relate to them. Hence, the change will affect the classes that use, inherit or call them other than themselves.
CCN	Changing the name of a Class will affect the class, its children and other classes that use it.
APC	Adding new *Parent classes* will affect all the classes that use or invoke their members and inherit from them.
DPC	Deleting existing *Parent classes* will impact all the classes that use, inherit or invoke them.
MPC	When an existing super class is changed, it will affect the derived class and other classes that are related to it.
AAbC	Adding the *Abstract Modifier* to classes will have an impact on all classes and subclasses that inherit or invoke or use members of the classes.
DAbC	Deleting the *Abstract Modifier* from classes will have impact on all classes that and subclasses that inherits, invokes and use members of the classes.
AFC	Adding the *Final Modifier* to classes signifies that those classes cannot be inherited and will no longer be visible to any related classes and subclasses. Thus, the change will affect them.
DFC	Deleting the *Final Modifier* from classes renders those classes inheritable and available to any classes. Thus, it will have no effect at all on existing classes related.

ASC Adding the *Static Modifier* to classes will impact all related classes, member classes and subclasses that use, call and inherit from them.

DSC Deleting the *Static Modifier* from classes will impact all related classes, member classes, and subclasses relating to them.

Change in Class Inheritance Derivative:

CID1 Changing *inheritance type* from *private* to *public* means all the public/protected members of the super classes will become the public/protected members of the subclasses respectively. Thus, no impact is created on any class.

CID2 Changing *inheritance type* from *public* to *private* means all the public/protected members of the super classes will becomes private members of the subclasses. Thus, the members are no longer visible to other classes and subclasses that invoke, use, or inherit them.

5.4.1.4 Package Change Impact Diffusion Range

In the context of this research, the type of change that can be performed on the package level that will have an impact on the execution of the OO software system is presented in this section. The abbreviation on the left hand side represents the package change type while the ***IDff*** description is on the right hand side.

Change Impact Range Description

APk Adding *new package* will have no impact on any other classes or packages since the existing classes or packages are yet to use them.

DPk Deleting an *existing package* means that the package will no longer be available to classes and other packages that reference it. Thus, all the classes and packages related to the deleted package will be impacted by the change.

MPkN Changing the name of a package will affect classes and other packages that use it.

5.4.2 OOComDN-1 Reachability

The reachability of the **OOComDN-1** is to assist the software maintainer with a *look-up table* to perform CIA on OO program involving a certain change type, *chtype*. The reachability of OOComDN-1, given by ***Reach,*** entails the ability to move from one vertex in a directed graph to some other vertex. On OOComDN-1, it is used in identifying components in the network which could be impacted if a change is effected on a particular component.

Definition 5.6: *[OOComDN-1 Reachability (Reach $<n_i,n_j,d^{type}>$]*

For a given OO program, ***P*** *and the equivalent* ***OOComDN*** $< N, D^E, D^{type} >$, *let* $n_i, n_j \in N$, $d^{type} \subseteq D^{type}$. *The Reachability* $<n_i,n_j, d^{type} >$ *is the transitive closure of its dependency,* $d^{type} \in D^E$ *which denotes that for all ordered pairs, there is a path from* $\boldsymbol{n_i}$ *to* $\boldsymbol{n_j}$ *on the OOComDN-1.*

Thus, in the context of this research Reachability is represented by:

$$\text{Reach} <n_i,n_j, d^{type} >$$

Where $n_i, n_j \in N$ and $d^{type} \in D^{type}$.

5.4.3 OOComDN-1 Look-up Table

The look-up table, **LT** is based on the OOComDN-1 reachability and is a table that contains the dependency types associated with a given type of change. In other words, it contains different dependency types involve in each change type that can be reached or checked during the course of impact analysis on a given component. That is, the impact diffusion of a change. The following definitions are given:

Definition 5.7: *[Look-Up Table (LT)]*

For a given OO program, ***P*** *and the equivalent* ***OOComDN-1*** $< N, D^E, D^{type} >$, *let* $n_i, n_j \in N$, $d^{type} \subseteq D^{type}$. *A Look-up Table, LT is a table that contains all* $\boldsymbol{D^{type}}$ that can be reached through **Reach** $<\boldsymbol{n_i,n_j, d^{type}} >$ *when a change,* ***chtype*** $\in$ *chset is considered on* $\boldsymbol{n_i}$ *on the OOComDN-1.*

The LT for each category of change based on their ***IDff*** category as follows:

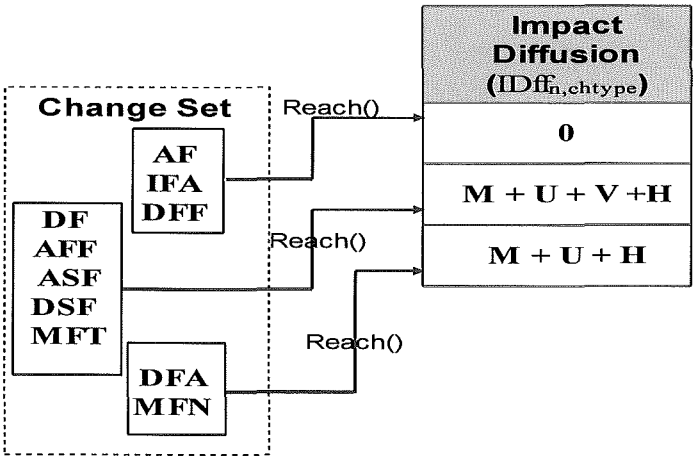


Figure 5.1: Impact Diffusion LT for Field Change

For instance, as shown in Figure 5.1, the change categories AF, IFA and DFF have the same ***IDff*** of 0, meaning no components will be impacted when a change is made with such *chtype*. This is also applicable to method and class LT.

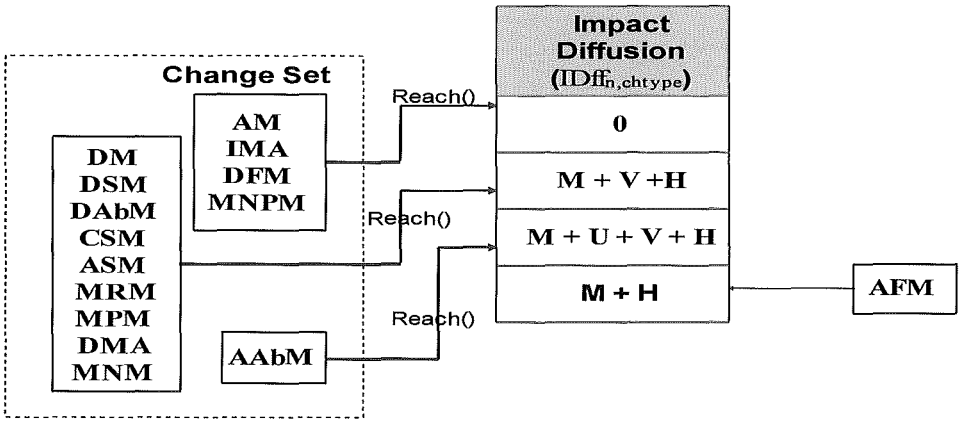


Figure 5.2: Impact Diffusion LT for Method Change

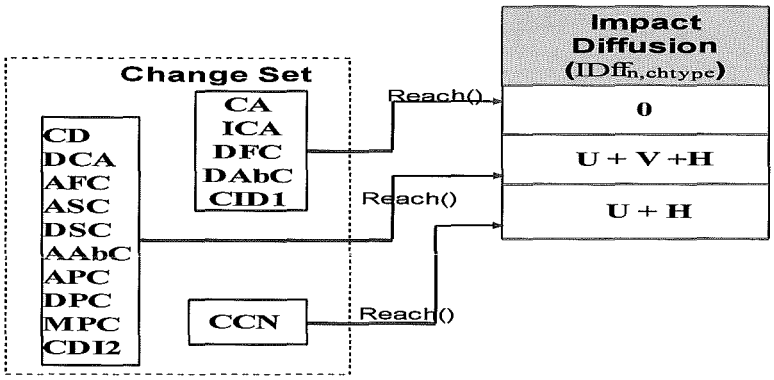


Figure 5.3: Impact Diffusion LT for Class Change

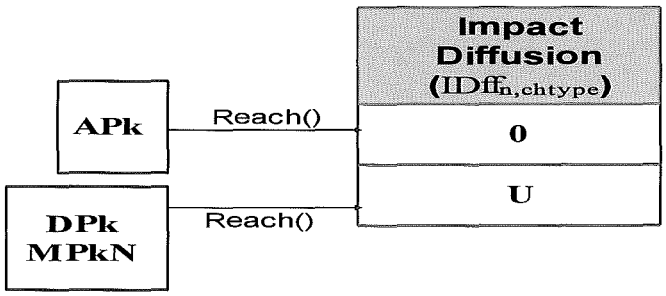


Figure 5.4: Impact Diffusion LT for Package Change

As shown in Figures 5.1, 5.2, 5.3 and 5.4, there are 44 kinds of ***chset*** and many are known to have the same ***IDff*** or expression representing the field, method, class and package respectively. The **LT** is a simplified representation of the different types of change ***IDff*** for OO software components. At the field level, $\mathbf{chset_F}$ = 10 and only 3 different ***IDff*** categories exist.

Accordingly, for member method and class, $\mathbf{chset_M}$ =15 and $\mathbf{chset_C}$ =16, 4 and 3 different ***IDff*** categories exist respectively. While in the package level, $\mathbf{chset_{Pk}}$ = 3 and only 2 different ***IDff*** categories exist. On the above LT, the connecting arrows labeled *Reach()*, denotes the dependency types, d^{type} that can be explored to identify the impact set when a component is changed with, *chtype*.

Hence, we give the formal definition of ***IDff*** as follows:

Definition 5.8: [Impact Diffusion ($IDff_{n,chtype,}d^{type}$)]

Given OO program, $\mathbf{P}^j$ *and its equivalent* ***OOComDN-1*** $< N, D_E, DT >$. *Assuming that an entity* e_i *and its corresponding vertex* n *on the OOComDN-1 is changed with chtype, where chtype*$\in$ *chset. Therefore, the impact diffusion range,* $IDff_{n,chtype}$ *is defined as follows:*

$$\text{IDff}_{n,\text{chtype},}\, d^{type} = \{\text{Reach} < \{\text{SIS}_n\}, n_{i,}\, d^{type}{}_{A\leftrightarrow B} >\} \quad \text{.....5.6}$$

where $d^{type}{}_{A\leftrightarrow B} \subseteq D^{type}$.

The expression in equation 5.6 denotes that the ***IDff*** is based on the ***SIS*** and the changed component and the impacted components are obtained through the bi-directional walk on the OOComDN-1 starting from the changed component, **A** to the impacted one, **B**. However, due to the strategy we have adopted in this research, in order to improve the precision and reduce the number of *impact set* generated by ***SIS***, the following impact rule is defined.

5.4.4 Impact Diffusion Rule

In the context of this research, the ***IDff*** rule is a rule that is used to complement a situation where the **SIS** violates the impact of a change and produces impact sets that are not truly affected by a change. In such a situation, the following rule will be applied.

If $\text{IDff}_{n,\text{chtype},}\, d^{type} = 0$;

Then,

$$\text{IDff}_{n,\text{chtype},}\, d^{type} = \text{IDff}_{n,\text{chtype},}\, d^{type} \cap \text{SIS}_n = 0 \quad \text{.....5.7}$$

The rule denotes that, for any change type such that $\text{IDff}_{n,\text{chtype},}\, d^{type} = 0$, and $\text{SIS}_n \neq 0$, $\text{IDff}_{n,\text{chtype},}\, d^{type}$ will depend on SIS through intersection rather than a union. By this, the number of false IS will be reduced or eliminated. For example, to use the *look-up tables*, LT let us for instance take a class ***chtype*** of the form, *chtype* = **CD**, deleting an entire existing, class with an equivalent node on the OOComDN-1, *n*, the SIS will be:

$$\text{SIS}_{c,\text{ CD},} = \{\text{CV}\}$$

Where CV is the change vector of the entity, *e* changed with ***chtype Є chset***. The CV for a class change is obtained directly from the matrix, m_{ij}. The expression in respect to SIS shows that deleting an entire class will impact the entities outside this class that are related to the class.
Thus, the impact diffusion for *chtype* = CD is:

$$IDff_{c,CD,\{U,V,H\}} = \{ \text{Reach} < \{SIS_{c,CD}\}, C_i, \{U + V + H\} > \}$$

The expression in respect to $\mathbf{IDff_{n,chtype,}}_{d}^{type}$ for ***chtype*** denotes that if an entire class, C is deleted, it will affect the whole class, the components in the derived classes of the class, and components in other classes that use or invoke the changed component.

5.4.5 Compound Changes

When a modification to object components involves several ***chtype*** simultaneously, it is called compound change. For example, a modification that involves changing the type of a field (MFT) and adding a *final* modifier (AFF) to the field at the same time. In such cases, the impact set is computed as a union for the ***chset*** involved in both **SIS** and *impact diffusion range*. This is achieved by performing each change one after the other and then computing the union as ***chset***. The rationale is to avoid complex situations that could affect the correct identification of the impact set. The definitions are as follows:

Definition 5.9: *[Compound Change]*
*Given OO program, P, let the component of P^j and the equivalent vertex, n on **OOComDN-1**. If this component involves several chset = {$chtype_1$, $chtype_2$... $chtype_n$}, and $chtype_i$ Є chset (i = 1, 2, ..., n); We therefore, obtained the SIS and impact diffusion range as the union of the chtypes on the component.*

Therefore, for SIS, component changed with chset ={$chtype_1$, $chtype_2$, ... $chtype_n$} is given by:

$$\mathbf{U_{i=1}^{m}\, SIS_{n,U_{n=1}^{m}\, chtype_i}} = U_{i=1}^{m}(\mathbf{CV_i}) \quad \text{.....5.8}$$

While the corresponding ***IDff*** expression based on the SIS is given by:

$$IDff_{n,U_{n=1}^{m}\, chtypei} = \{n_i | \text{ Reach.} < U_{i=1}^{m}\, SIS_{n,U_{i=1}^{m}\, chtype_i,}\ n_i, U_{i=1}^{m}\, dtype > \} \quad \text{.....5.9}$$

To have an insight into the above expressions, assuming we want to change the *type* of a field and add the modifier *final* to a class field at the same time, SIS and the ***IDff*** of these two *chtypes* will be computed as:

$$SIS_{F,\, U(MTF,ASF)} = \{ SIS_{F,MFT} \cup SIS_{F,ASF} \}$$

SIS is computed from the above discussed matrices, while the corresponding ***IDff*** for the $SIS_{F,U(MTF,ASF)}$ is given by:

$IDff_{F,\ \cup(MFT,ASF)} = \{ n_i \mid Reach < SIS_{F,\ \cup(MFT,\ ASF)}, F_i, \{M+U+V+H\} \cup \{ M+U+V+H \} > \}$

$\equiv IDff_{F,\ \cup(MFT,ASF)} = \{ n_i \mid Reach < SIS_{F,\ \cup(MFT,\ ASF)}, F_i, \{M+U+V+H\} > \}$

In summary, for CIA involving several different changes, the resultant *impact set* is simply calculated based on union principle. Thus, we can now give the definition of the EIS as follows:

Definition 5.11: *[Estimated Impact Set, $(EIS_{n,chtype})$]*

*Consider OO program, **P** and its resultant **OOComDN-1** $< N, D^E, D^{type} >$. Suppose a component e and its equivalent node n $\in$ N on OOComDN-1 is changed with chtype $\in$ chset. Therefore, the EIS becomes:*

$$EIS_{n,chtype} = \{IDff_{n,chtype,d}^{type}\}$$

For ease of comprehension, EIS is an in-depth expansion of the **SIS** and the expression shows that **EIS** depends on three influencing factors: the **SIS**, **dependency types**, d^{type} between components and the **change type(s)**, *chtype* of the modification. Consequently, it depends on the **impact diffusion range** of the type of change on the changed component. However, **SIS** and **EIS** ought not to be different and in such case, the impact is limited to what was originally thought to be modified.

5.5 Static Impact Prediction and Total Impact Set

Impact analysis is guided by the principle that ripple-effect is inevitable when a change is implemented on an OO program regardless of how small the change may be. Although, the affected parts are often not known or easily detected, CIA process is said to be iterative and discovery in nature, revealing more impacts while a change is being carried out. In previous sections, SIS and EIS based IDff of different *chtype* have been devised. In this section we will discuss TIS and AIS of the static impact analysis process of this research. The process is captured in Figure 5.5.

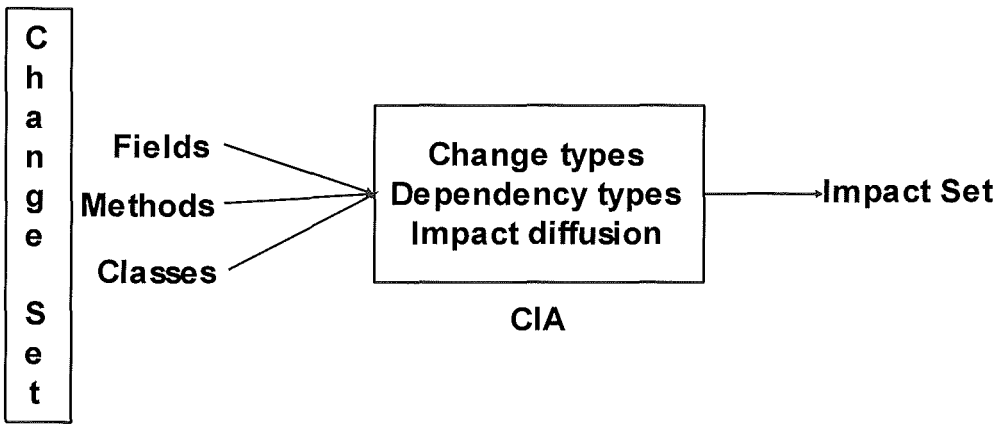


Figure 5.5: Static CIA Process

The process represented in Figure 5.5 accepts inputs in the form of software such as packages, classes, methods and fields that require a change. CIA is the technique that processes the change sets while taking into account: the different *chtype*, d^{type} and the *IDff* of the *chtype*. Furthermore, impact set is the resulting end product (output) of the process and it originates from the **SIS**, **EIS** and **TIS**.

Definition 5.12: *[Total Impact Set ($TIS_{n,chtype}$)]*

TIS is the set of all components thought to be affected by change during the CIA process. In this case, TIS can be obtained by computing the union of ***SIS*** *and EIS as follows:*

$$TIS_{n,chtype} = \{\sum_{i=1}^{m} (SIS_n + EIS_{n,chtype})\} \quad \text{.....5.11}$$

In some cases, **TIS** is equivalent to **EIS**.

Definition 5.13: *[Actual Impact Set ($AIS_{n,chtype}$)]*

AIS is the set of components truly affected by a change and modified when the change is carried out. AIS can be obtained by computing the union of SIS and EIS as follows:

$$AIS_{n,chtype} = \{TIS + True\ IS\} - False\ IS \quad \text{.....5.12}$$

In equation 5.12 ***True IS*** is the set that was truly impacted but not predicted by the CIA technique while the ***False IS*** are the IS predicted but not truly affected by the change. The best-case scenario of this computation as stated by [5], is a situation where TIS and the AIS are alike, signifying that the impact prediction was perfect at the level of granularity employed: package, class and class member level.

5.6 Change Proposal and Criteria Representation

A change proposal (CP) is a change that a software maintainer or developer proposes to realize in the existing software. It identifies the set of documents defined at high-level considered to be candidates for SIS. This thesis assumes that a CP has already been expressed in terms of source code components that will be changed. The components are called *targets* of the CP. Every given CP is guided by a set of change criteria, CC and the change impact for each CC that will be computed. In other words, CC is what needs to be done to discover the impact set of a change.

Definition 5.14: *[Change Criteria (CC)]*

For every change, we define CC as $<C_{(Pk,C,M,F)}, ch, IDff_{n,chtype,d}^{type}>$, *where* $C_{(Pk,C,M,F)}$ *specifies the affected component (i.e. package, class, method or field) proposed to be changed, ch is the possible change type, and* $IDff_{n,chtype,d}^{type}$ *is the IDff or expressions of each chtype identified.*

$$CC = <C_{(C,M,F)}, \text{chtype}, IDff_{n,chtype,d}^{type} > \quad \text{.....5.13}$$

For instance, to delete an entire class with a corresponding node *n* on the OOComDN-1, the CC will be:

$$CC = <C_{(C)}, CD, \text{IDff}_{n,cd,\{U,V,H\}}>$$

5.7 Typical Illustration

Before illustrating a simple example of CIA technique discussed in this chapter, let us outline all the steps that are required to carry out CIA when a CP is considered.

Steps:

1) Firstly, analyze the original OO program (e.g. Java) based on the proposed change and construct the OOComDN (OComDN-1 and OOComDN-2). We assume the software has been thoroughly tested.
2) Transform the OOComDN-1 into $[m_{ij}]$, $[k_{ij}]$ and $[p_{ij}]$ matrices to show both high-level and low-level dependencies respectively.
3) Identify the proposed changes based on the code change type categorization of: *packages, classes, methods and fields.*
4) After obtaining the different change types categorization applicable:
   i. Form the change criteria, CC for each affected component
   ii. Compute the union of the different types of change for the component proposed to be modified.

$$\textbf{chset} = \textbf{chset}_{Pk} + \textbf{chset}_{C} + \textbf{chset}_{M} + \textbf{chset}_{F}.$$

5) Based on the CC, compute SIS for each CC. Thus, SIS can be computed as follows:
   i. Compute the CV for each change entity in $[m_{ij}]$, $[k_{ij}]$ and $[p_{ij}]$ where applicable.
   ii. Compute the union of all CV if any as the SIS.
6) After obtaining the SIS, compute the EIS of each components based on SIS using the ***IDffs*** of the identified *chtype,* and
7) Finally, compute the **TIS** as a union of the SIS and the EIS.

Based on the above steps, below is a demonstration of the method using a sample java program shown in Figure 4.1 of chapter 4. The CP for this example is captured in Table 5.1 with following columns:

i. Column 1 contains the change ID
ii. Column 2 is the CP containing candidate component before and after a change is implemented, and
iii. Column 3 is the code change types applicable.

The CIA process steps discussed above are performed as follows:

STEP 1:

Using OOComDN-1 and the matrices in Table 4.3, and Figure 4.5 and 4.6 respectively, the first step is to analyze the CP to identify the various change types involved as shown in Table 5.1.

Table 5.1: Change Demonstration

Change ID	CP		Components Change Types (ch_i)
	Components Before Change	Components After Change	
1	private *int* **d**	public string d = "2"	*IFA, MFT*
2	Public void M3()	Abstract public void M3()	*AAbM*
3	Public class A	Public class E	*MNC*

STEP 2:

Based on the different source code change types for the three changes extracted from CP in Table 5.1, the next step is to form the individual CC.

$CC = <C_{(C,M,F)}, chtype, IDff_{n,chtype,d}^{type}>$, such that :

1. $CC = <C_{(d)}, (IFA,MFT), IDff_{d,U(IFA,MFT),(0,M,H,V,U)}>$
2. $CC = <C_{(M3())}, (AAbM), IDFF_{M3(),AAbM,(M,H,V,U)}>$
3. $CC = <C_{(A)}), (CCN), IDff_{A,CCN,(U,H)}>$

Hence, the overall change set for all the changes to be performed on the program, P^j are:

chest = {IFA + MFT + AbbM + MNC}

STEP 3:

In this step, we have to compute the SIS using the change vector, CV as follows:

1. $CC_1 = <C_{(d)}, (IFA,MFT), IDff_{d,U(IFA,MFT),(0,M,H,V,U)}>$

Since the corresponding vertex *n* is ***"d"*** which matches up with the vertex, A in the OOComDN-1, we will therefore use the intra-membership relation matrix of A. CV is derived via column-row basis in the matrix.

A_i/A_j	d	A()	M1()	M2()
d	1	0	0	0
A()	0	1	0	0
M1()	1	0	1	0
M2()	1	0	1	1

Figure 5.6: Intra-membership Matrix of A

A_i/D_j	d	A()	M1()	M2()
q	0	0	0	0
D()	0	0	0	0
M6()	0	0	1	0

Figure 5.7: Inter-membership Matrix of A and D

As shown in Figure 5.6, the CV corresponding to vertex ***"d"*** in A is:

CV=<1,0,1,1> =<d, **M1(), M2**> = $SIS_{d,IFA}$

$IDff_{d,IFA,d}^{type} = 0$;

But applying equation (5.7) rule

$$IDff_{d,IFA,0} = IDff_{d,IFA,0} \cap SIS_d = 0$$

Therefore,

$$\mathbf{EIS_{d,IFA}} = \{IDff_{d,IFA}\} = \{\phi\}$$

Also, as shown in Figure 5.7, the CV corresponding to vertex ***"d"*** in D is:

CV = <0,0,1> <d, **M1(), M2**>

Thus, SIS = {M6()}

But using the LT, we have $IDff_{d,MFT,(M,H,V,U)}$ = {**d, M1(),M2()**}

$$\rightarrow \mathbf{EIS_{d,MFT}} = \{IDff\} = \{d, \mathbf{M1(),M2()}\}$$

Thus,

$\mathbf{TIS_{d,MFT}}$ = { $SIS_{d,MFT}$ } **U** {EIS} = {d, **M1(),M2**(),M6()}.

As shown in the above computation, for the change type, ***chtype*** = IFA, SIS=EIS=TIS=0 for *"d"* meaning it has no impact on any component within and outside the class. In the same vein, the change type, chtype = MFT is TIS = {d, **M1(),M2**()} for *"d"*., meaning only ***d, m1() m2()*** and ***m6()*** in A will be affected when changes of the type MFT is implemented.

2. $CC_2 = <C_{(M3())}, (AAbM), IDFF_{M3(),AAbM,(M,H,V,U)}>$

With this change, since the corresponding vertex *n, on OOComDN-1* ***"M3()"*** that matched up with the vertex n, **B** in the OOComDN, we will therefore use the intra-membership relation matrix of **B** to compute our SIS.

B_i/B_j	a	B()	M3()	M4()
a	1	0	0	0
B()	0	1	0	0
M3()	1	0	1	0
M4()	1	0	1	1

Figure 5.8: Intra-membership Matrix for B

Figure 5.8 is the intra-membership relation matrix of B. In Figure 5.8, the CV corresponding to vertex **"M3()"** in B is:

CV=<0,1,1> =<**M3(), M4()**>

Hence, the $\mathbf{SIS}_{M3(),AAbM}$ = {**M3 (), M4()**}.

But using the LT, $IDFF_{M3(),AAbM,(M,H,V,U)}$ = { B, M3(), M5(),M6}

→ $\mathbf{EIS}_{M3(),AAbM}$= {IDff} = {M3(), B, M5(),M6}

Thus,

$\mathbf{TIS}_{M3(),AAbM}$= { $SIS_{M3(),AAbM}$ } **U** {EIS} = {B, M3(), M4(),M5(), M6()}

The result shows that, if a change is performed on M3() of the type, *chtype* = AAbM, components B, M3(), M4(), M5(), and M6() as shown on the OOComDN will be impacted.

3. CC_3 =<$C_{(A)}$, (CCN), $IDff_{A,CCN,(U,H)}$ >

Since the corresponding vertex *n* is **"A"** which matches up with the vertex **A** in the OOComDN-1, we will therefore use the class dependency matrix of the entire program, P.

	A	B	C	D
A	-	o	o	o
B	+	-	o	o
C	o	+	-	o
D	+	+	+	-

Figure 5.9: Class Dependency Matrix of A,B,C and D

Figure 5.9 is the CDM for classes A, B, C and D. The CV corresponding vertex **"A"** in **A** is:

CV=<1,1,0,1> =<**A, B, D**> Where "-"= "+"=1.

Hence, the $SIS_{A,CCN}$ = {**A, B, D**}.

But if we use the LT, $IDff_{A,CCN,(U,H)}$ = {A(),B, D, M6()}

The $EIS_{A,CCN}$ = {IDff} = {A(), B, M6() }

$TIS_{A,CCN}$ = { $SIS_{A,CCN}$ } **U** {EIS} = {A(), B, D, M6()}

Here, the result computed shows that if a change of the type, chtype = CCN is implemented in a class, A, It would impact components such as A(), B, D, and M6().

In general, the TIS for the entire *chset* for the CP are given as follows:

$$\mathbf{TIS} = \sum \mathbf{SIS}_{\{d,M3(),A\}} + \mathbf{EIS}_{\{d,M3(),A\}} = \{M1(), M2(), M4(), M6, B, D\}$$

$$\mathbf{EIS}_{\{d,M3(),A\} U (IFA,MFT,AAbM,MNC)} = \{\mathbf{d}, \mathbf{M1()}, \mathbf{M2()}, M3(), M4(), M5(), M6(), A(), B, D\}$$

5.8 Metrics for Evaluating CIA Techniques Effectiveness

Evaluating the effectiveness of any technique in software process is essential for knowing if the process is effective at what is designed for. In the perspective of CIA, metrics are used to evaluate the process as soon as the changes have been made. This is done by comparing the predicted impact set TIS with the AIS in order to know the degree of discrepancy in the process. Measurement of this nature is necessary for analysis, learning and continuous improvement of the process capability. Based on the work of [12][66], two metrics are defined for the effectiveness evaluation of the CIA technique. Precision and recall are the basic measures employed in the area of information retrieval for evaluating search strategies, but they are also important in the context of impact analysis [12][59][66]. As an important objective in this research, we aim at reducing the number of *false-positive* (high **precision)** and *false-negative* (high **recall**) as much as possible [59][66]. These two phrases originate in the following context.

- If the analysis results obtained contain specific components not truly affected by a change, thus an error was made and is referred to as a *false-positive*.
- On the other hand, when the analysis results do not contain truly affected components, an error was made and is known as a *false-negative*.

Thus, the metrics we chose for evaluating our CIA techniques in this research are defined as follows:

Definition 5.15: *[Precision]*

Given the EIS and the AIS, ***Precision*** *is defined as the ratio between correctly estimated components and the total of estimated components. In other words,* ***Precision*** *measures how many of the OO software components predicted as impact set actually have been shown to be truly impacted. Mathematically, we express* ***Precision*** *as:*

Let AS be the AIS, and ES represents the EIS. Precision is therefore:

$$\text{Precision} = \frac{AS \cap ES}{ES} \; x \; 100\% \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 5.14$$

Based on our CIA approach, ES is obtained when performing the impact analysis which is simply computed from the union of the SIS and the impact diffusion range of the change type involved. With this equation, a high ***Precision*** indicates a low number of *false positives* and a low ***Precision*** means a high number of *false positives* which is what this research tends to avoid.

Definition 5.16: *[Recall]*

Given the EIS and the AIS, we define ***Recall*** *as the ratio between correctly estimated components and the total of impacted components. In other words,* ***Recall*** *measures how many of the impacted OO components are actually predicted as impact set. Mathematically, we express* ***Recall*** *as:*

Let **AS** be the AIS, and **ES** represents the EIS. Recall is therefore:

$$\text{Recall} = \frac{\text{AS} \cap \text{ES}}{\text{AS}} \; x \; 100\% \quad \text{.....5.15}$$

A high ***Recall*** indicates a low number of *false negatives* and a low ***Recall*** means a high number of *false negatives* which is unacceptable in this research. There is a trade-off that exists between precision and recall. This trade-off stems from the fact that high ***precision*** result to a reduction in ***recall*** and a high ***recall*** results in a reduction in ***precision***. Based on the above equation, they are situations in which ES and/or AS is 0. Such situations can be dealt with as follows:

If **ES** = 0, ***Precision*** = 100%, and

If **AS** = 0, ***Recall*** = 100%.

5.9 Experimental Analysis

In this section, we present the result of the case study conducted in the context of this thesis. The goal is to evaluate the effectiveness of the CIA technique we developed in this study which aimed at answering research questions **RQ1:2** and **RQ1:3** of **MRQ1**. The description of the case study is discussed as follows:

5.9.1 Study Systems

This case study utilized systems developed by the same subjects discussed in Section **4.5.2** of chapter 4. In this case, the subjects developed small to medium-sized systems of their choice in their second semester project. In this project, the students were given two months to develop their system. About 55 students participated but after careful consideration based on the quality of the systems built, only 50 students' projects were considered for this analysis. In all, we had ten teams of 5 students each. In this project, no restriction was imposed unlike the project used for the above experiment. The students were asked to develop the system using java programming language and they followed any software development model of their choice. At the completion of the project,

each team submitted workable system's source code with a maximum LOC of about 200 arranged into packages, classes, methods and field. In addition, they also provided a full documentation of their work. Each system submitted was thoroughly acceptance tested before being graded by the lecturer responsible for the course.

5.9.2 Project Characteristics

Each project was graded, and the following features were collected, as shown in Table 5.2. The first column is the team, followed by the number of packages, classes, methods and lines of code.

Table 5.2: Study Project Characteristics

Team	Package	Classes	Methods	LOC
A	2	5	10	102
B	3	7	15	152
C	2	6	13	125
D	2	5	9	98
E	3	6	12	110
F	2	5	10	105
G	2	6	11	119
H	2	5	10	109
I	3	7	14	145
J	2	6	12	117

5.9.3 Analysis and Results

In this study, the goal was to measure the effectiveness of the CIA technique developed to see if it actually improves the precision of the impact set. For each system, we carried out CIA on different change requests that reflect the different type of changes applicable for OO program. (see Section 3.3 of chapter 3) Due to the size of the systems studied, we performed CIA based on the following change types:

Change category		**Change Type**
Package change	–	Modify the name of an existing package
Class change	–	Add a new empty class
Methods change	–	Increase a method access from private to public
		Delete existing method and
Field change	–	Decease field access from "public" to "private"

We carried out CIA on the systems based on impact diffusion of the type of OO program change. In this case, we followed strictly the steps discussed in Section 5.7 which begins with the construction of the IR called OOComDN-1 for each of the programs written in java. Each edge corresponds to the dependencies types while the nodes corresponding to the changed component (package, class, method or field) and other nodes were inspected for any impact. OOComDN-1 was then transformed into $[m_{ij}]$, $[k_{ij}]$ and $[p_{ij}]$ matrices. Based on this, the SIS was computed

based on the adjacency matrix while EIS was computed based on SIS using the *IDffs* of the identified *chtype*, and finally, the TIS as a union of the SIS and the EIS was computed.

To measure the effectiveness of the CIA technique if precision was actually improved, we used the metrics discussed in Section 5.8, ***Precision*** and ***Recall*** which measure the numbers of false-positives and false-negatives respectively. The objective is to reduce the number of false-positive (high precision) and false-negative (high recall) as much as possible. The precisions and recalls of the TIS for all the 10 projects are captured in Figure 5.10.

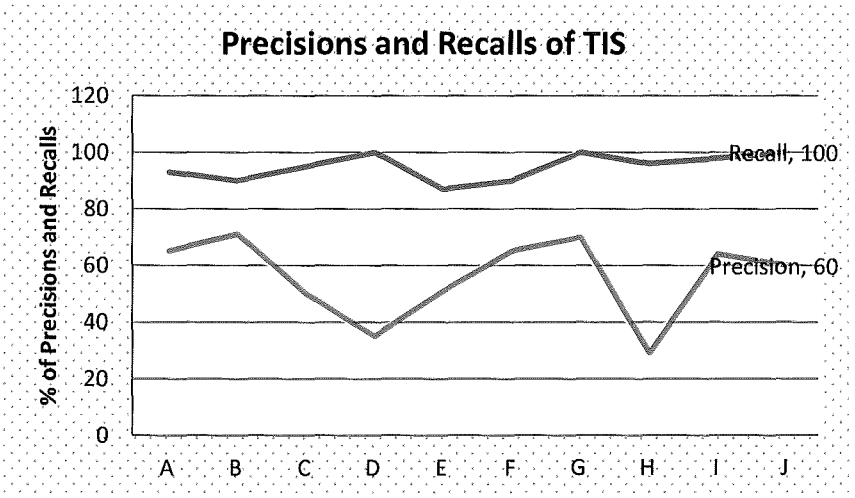


Figure 5.10: Percentage Precisions and Recalls

As shown in Figure 5.10, it can be seen that the recalls of the total impact set are high. This is the case of any static CIA technique. We achieved a high recall of 100% in three of the projects indicating a low number of false negatives. Furthermore, the precisions of the total impact set also improved as expected. We had precisions of between 29 and 71%. This result however, indicates low number of false positives. When compared to the work of [12], we found that the recalls were similar but our precisions improved by 11%. Based on the systems' characteristics shown in Table 5.2 and the results in Figure 5.10, it is quite obvious that our CIA technique is effective and can be used in teaching undergraduate students OO software maintenance as modification is only based on the type of change, the dependencies types, the impact diffusion and not the size of the software application. Using this technique in larger application could be effective as well.

5.9.4 Limitations

There are several threats to the validity of the results presented in this case study. In this case study, we have used students' projects which are characterized as small or medium-sized systems for evaluating our CIA technique. In this case, generalization can only be done when larger applications are used. Though the programs are students' projects, they are of good quality, real

and appropriate for this study. In addition, we have only used precision and recall to evaluate the effectiveness of the CIA technique. There are other measures used in similar studies that were not used such as F-measure [12]. Thus caution is needed to interpret the results. However, we are confident the measures we used are appropriate for this study.

5.10 Chapter Summary

In this chapter we have discussed the various approaches for predicting the ripple-effect of a change on different OO program components that have dependencies on the changed component. To achieve this, there is an impact technique that is based on the different change categories of OO software, the type of dependencies that exist between the changed and other components as well as the impact diffusion range of each change type. The approach shows that, performing changes on OO software by taking those influencing factors into consideration constitutes a giant step towards successful maintenance while improving the precision of the impact set's predictions. Two measures were also discussed which can be used to evaluate the effectiveness of the approach in order to assist in taking good decisions. The results of the evaluation indicate that using the IR alongside the CIA techniques improves the precisions of the impact set.

CHAPTER 6

Fault-Proneness Measures

6.1 Introduction

This chapter focuses on software measures in the context of maintenance and fault-proneness. The goal is to support maintenance activities in large-scale OO programs by improving the quality of the product before and after change through software metrics. It begins with general background information of software change risks and the methodology used, and progresses to measures which are related to fault-proneness of a class. Furthermore, we discuss the management of change information for effective collection during software failure prediction.

6.1.1 Background Information

Due to the increase in size and complexity of software applications today, quality has been a key concern even since the initiation of computer software. For a developed software product to be of high quality, development and testing methodologies need improvements that are continuous, leading to spending fewer resources, effectual test cases, and better coverage [87][105]. However, assuring high quality is an increasingly complex time and effort-consuming activity [105]. As we know, change is an indispensable property of any software which is necessary to keep the system useful during its lifetime. Yet, each change made to a software system carries with it some possibility of failure that could manifest either during testing or in the field. This failure could be due to two factors: *negligence of dependencies between the software components* and *process activities such as the frequency of changes, number of developers performing the changes, their maintenance experiences, the file age*, and so on. The first case is related to the structural properties of the software products and can be eliminated or minimized through effective CIA activities while the latter cannot be eliminated through CIA activities alone. This means better and more effective ways of eliminating them early in order to avoid costly field failure which could impact the product users negatively should be in place. In the realm of OO programs, classes are the basic units of analysis, and thus their quality is critical to the overall quality of the software. As faults in a software system may lead to failure in an executable product, committing a change on a faulty class without a pre-knowledge of such fault may result in a software failure or increase the risk of field failure.

In large and complex software systems, one common issue is the existence of faults, though recent evidence shows that most of them are only found in a few system components [9]. However, preventing or reducing the number of software failures is one difficult task. Several empirical

studies have shown that, if faults cannot be fully prevented, the best and most effective thing to do is to try find and remove them early. Although software testing, code inspection, and walkthrough methods can find software faults, and debugging alongside testing can remove them [105], software testing is seen as very expensive with respect to time and resources. One good approach is to measure the software product quantitatively and use the results to predict the potential faults that are likely to result in failure. Identifying such components early would allow mitigating actions to be focused on the high risk components which are likely to cause field failures. The software tester's work can be made more effective and efficient. In this thesis, one key objective is to avoid risky and costly software changes that are embedded with the probability of failing in testing or in the field. To achieve this, the task is to predict the probability of failure early during CIA using software metrics that are known to have influence on fault-proneness.

The maintenance of large systems today is very difficult. This stems from the fact that, for instance, performing *after-release* modifications does not only require a careful understanding of the *structural design*, *dependencies* and *interactions* of the software to be changed but also the risk assessment of the impact of such change fixes. Before implementing a proposed change, efforts should first be channeled towards identifying which classes are likely to be faulty or result in field failure if changes are implemented on them. This will proffer an effective technique for improving the quality of the product. This thesis will therefore use the static structural code measures of a class in the present release, **R** and faults as well as change history of the class from past release, **R-1** to predict the fault-proneness of a class.

6.2 Software Measures

Software metrics is used to evaluate the quality and complexity of a software as well as to take useful managerial and technical decisions related to cost, effort, time, quality and so on [41][44]. In the context of this research, we focus on collecting quality measures that are important in predicting the fault-proneness of an OO class. In this case, *product-related* metrics that measures the structural properties of OO classes and *process-related* measures based on the change characteristics and fault history of a class are used.

6.2.1 Software Fault-Proneness

As quality is the ultimate goal in Software Engineering, one important research area in software metrics is identifying the connections between *internal* and *external quality attributes*. This relationship is best established statistically through correlations between such internal attributes and fault-proneness, in particular. A substantial amount of research energy has been devoted to defining particular quality measures that are used to build quality models which in turn aid

decision-making during software development. The fault-proneness of classes has been the most frequently used software measure in the capacity of dependent variable [87][107]. This stems from the assumption that a class under development is fault-prone if it has the same or similar properties defined by software measure as a faulty class in a previous release, **R** in the same environment. Accordingly, if we can accurately estimate the occurrence of faults in the software early, efforts and resources can be channeled to the more fault-prone parts. Many software metrics have been proposed and several empirical studies have studied the relationship between them and fault-proneness [11][41[87]. These studies were geared towards demonstrating the validity of software measures in order to continue to remain useful in the industry.

This thesis therefore wants to predict the fault-proneness of classes during CIA in order to focus validation and verification efforts on such classes before and after implementing the change. However, the main question is *"which of these metrics are useful in measuring the fault-proneness of classes?"* Identifying the measures which are predictors of fault or failure-proneness will go a long way to answering **MRQ2** → *RQ2.1*. To achieve this, we have used a SLR and CLR. The methodologies are discussed in the following subsections.

6.3 Sub-Research Methodologies

This section gives brief summaries of the methodologies employed to collect software metrics which are used as predictors of fault-proneness. We used SLR to collect *OO design metrics* and size measure, while CLR was used to collect *process-related* measures: fault data and change history of a software project. In both cases, we conducted the review to have an in-depth understanding of the state-of-the-art in fault-proneness predictions and measures used since several product and process metrics are presently in existence. We adopted these methodologies (SLR and CLR) because we lack the resources to carry out empirical studies on large-scale real-world software systems in order to collect metrics. In addition, the objective was to help software engineers in taking quick decision as to which metrics are generic for class fault prediction when OO design, size and change measures are used.

6.3.1 The Systematic Literature Review

This section summarizes the SLR performed in this thesis, though the comprehensive report can be found in [18] by Isong et al. Ideally, SLR as used in the field of medicine and applied in software provides the means to identify, evaluate and interpret collection of relevant published research findings which assist in answering stated research questions [106]. As stated by Kitchenham et al. [106], SLR is a form of secondary study while individual studies that add to it are the primary studies. SLR uses a well-defined methodology that is considered unbiased. In this case, this thesis followed strictly the guidelines recommended by Kitchenham [106] which are

planning, conducting and reporting the review. In planning, the key activities we performed were the definition of the research questions and the review protocol development. Table 6.1 provides a mapping of the research questions, review questions and methodology used.

Table 6.1 Mapping of Research Questions, Review Questions and Methodology

Research Question (MRQ2)	Review Questions	Methodology	Steps
RQ2.1. Which metrics are suitable for OO program's fault prediction?	RwQ1: Which metric (s) among CK metric suit and SLOC has impact on FP of a class?	SLR	Explore empirical studies that have validated CK +SLOC metrics on real-world application to find which are significant or not.
	RwQ2: What techniques are being used to validate the metrics in R1 and which is the best?	SLR	Explore the empirical evidence to find out which techniques are used and the best among them.
	RwQ3: To what extent have the metrics in RQ1 been validated?	SLR	Find out from the studies, the systems used in the validation, the environment, and the programming languages and so on.
	RwQ4: Of what relevance are the empirical validations?	SLR	Explore why the validations of these metrics is important
	RwQ5: Are there generic metrics for predicting faulty classes?	SLR	Find out if certain metrics can be considered generic for fault prediction.
RQ2.2 Based on the metrics, how can we formulate a model that predicts the early failure of components which are affected by change request?	RwQ2	SLR	Same as RwQ2

In the conduction stage, the relevant primary studies were selected according to the defined search strategy and the data were extracted and synthesized. In the SLR, the study considered the review of 17-years' efforts in empirical validation of CK+ SLOC metrics, between the period of January 1995 to December 2012. The studies were chosen with respect to CK+SLOC metrics based on their level of significance with fault-proneness, their state of validation, and their usefulness in terms of software quality. A number of relevant article sources were used as shown in Table 6.2.

Table 6.2 Databases Used

Database Name
Google Scholar
Engineering Village (Compendex, Inspec.)
Scopus

In the SLR, twenty-nine (29) empirical studies were found to be relevant and the quality assessment of the primary studies was also performed. The process we adopted is shown in Figure 6.1.

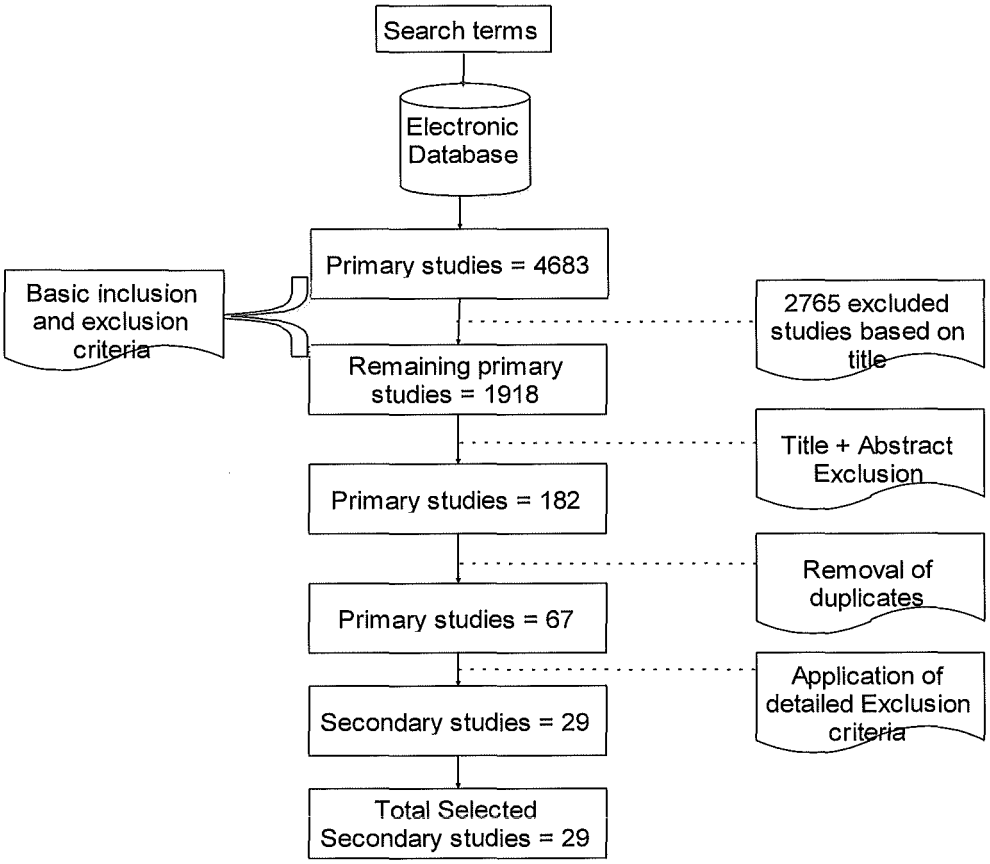


Figure 6.1: SLR Process

Finally, the findings of the SLR were documented, analyzed and qualitatively presented. Analysis of the findings is presented in chapters 6 and 7 of this thesis. Hence, incorporating SLR into this research is important in obtaining empirical evidence that provides information on the state-of-the-art of the impact of software metrics on fault-proneness of OO classes and to reduce research bias.

6.3.2 Comprehensive Literature Review

Unlike SLR, CLR is a commonly used methodology by researchers and practitioners to discover the existing knowledge on a subject area. It can be described as progressive steps of collecting, knowing, understanding, applying, analyzing, synthesizing, and evaluating the quality of the literature so that strong grounds for a topic or research method can be established [52]. Approaches used for performing CLR can differ since it depends on the objective and scope of the research. Nonetheless, it should be carried out in an effective manner. In the context of this study, we performed CLR on several empirical studies that have validated the relationship of different process-related measures with fault-prone classes. In these empirical studies, *measures of change history* and *fault data* were used, which are obtainable from the change management activities and

fault tracking databases. The outcome of this review is a set of metrics which are good predictors of fault-proneness.

6.4 Product Measures

Product measures are measures that can be computed directly from the software under study. They are known as the code attributes of the software. The product metrics considered in this thesis are the static measures that include SLOC and OO design metrics which capture the structural properties of OO programs. The choice of these metrics is based on the fact that several empirical studies have shown that these measures have a strong relationship with fault-proneness of a class [107][115]. Other measures not considered here which have been used in other related studies are *McCabe's cyclomatic complexity*, *degree of statement nesting*, *Halstead's program volume* [115] and so on. We do not consider them because the measures do not support certain key OO concepts like classes, inheritance, encapsulation and message passing [41]. We give analysis of the measures based on the SLR we conducted as follows:

6.4.1 Lines of Code and OO Complexity Metrics

Unlike the procedural programs, certain structural features of OO programs have been implicated in reducing its understandability and increasing its cognitive complexity leading to its fault-proneness. Several metrics have been proposed to capture the structural quality of OO code and design. Among these metrics, a commonly used set of OO metrics adopted is the one proposed by [41], also known as CK metric suit. CK metrics display the measurable features of OO software, and raised huge interest among researchers and engineers, and triggered many empirical studies that validated those metrics. CK metric suits are the foundation of OO metrics since others are based on them.

In the SLR conducted in [18], the findings showed that some OO metrics are good predictors of fault-proneness. Analysis indicates that coupling, complexity and size measures have strong impact on fault-proneness of OO classes, though the findings from some studies were not consistent. For instance, a metric considered significant to fault-proneness in one study might be insignificant in another study. This can impede decision making in selecting directly the measures that are related to class fault-proneness. However, the question still remains *which of these metrics are useful in capturing fault-proneness of OO classes?*

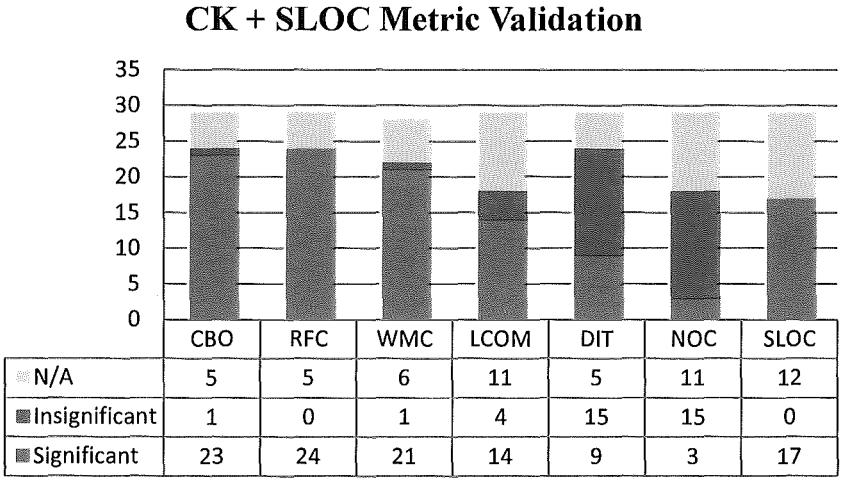


Figure 6.2: Validation of CK + SLOC Relationship with Fault-proneness

The metrics that are significantly or insignificantly related to fault-proneness are captured in Figure 6.2. Accordingly, Table 6.3 presents the different studies that have found CK and SLOC metrics to be significant and insignificant. In most of the studies we considered, the terms strong, weak, positive, and negative were used to quantify the degree of the relationships, while most studies considered faults in a class based on the level of severity such as high, medium, low and ungraded [95]. But in this study, no distinction between these terms was made, rather we focused only on whether a metric is significant or not.

Table 6.3 OO and SLOC Metrics Validation

Metric	Significant	Insignificant	N/A
WMC	[80][81][53][82][11][86][87][88][89][82][90] [91][92][93][94][95][97][99][100][103] [104]	[82]	[45][83][84][85][98][101]
LCOM	[80][83][84][86][45][90][91][92][93][19] [95][100][101][104]	[88][40][97][98]	[81][53][82][11][85][87][89] [96][98][102][103]
CBO	[80][81][82][83][11][86][87][45][88][89][40] [90][91][92][93][94][95][97][98][99] [100][103][104]	[53]	[84][85][96][101][102]
RFC	[80][81][53][83][11][85][86][87][45][88][89] [40][90][91][92][93][94][95][97][98] [99][100][103][104]	-	[82][84][96][101][102]
DIT	[82][83][85][45][89][40][93][95][104]	[80][53][11][86][87][88][90] [91][92][94][97][98] [99][100][103]	[81][84][88][101][102]
NOC	[80][89][91]	[86][87][45][90][91][92][93][94][95][97][98] [99][100][103][104]	[81][82][83][84][11][85][88] [40][96][101][102]
SLOC	[80][81][82][86][89][91][92][93][94][95][96][97][98][99][100][102][103]	-	[53][84][11][85][87][45][88] [40][90][101][104]

**N/A: not applicable

In all the 29 studies that have validated the relationship between CK + SLOC metrics with fault-proneness of a class, the findings on individual metrics are given as follows.

1. ***Complexity Measures:*** In the SLR, analysis shows that 21 studies confirmed WMC metrics impact on fault-proneness of OO classes. The finding indicates that OO classes that have more member functions or methods are more likely to have faults than classes with small or no member functions. However, only one study found it to be insignificant and 6 others studies did not measure it at all.
2. ***Coupling Measures:*** In the perspective of coupling measures, analysis indicates that 23 of the studies found that CBO had a strong influence on class fault-proneness. The significance stems from the fact that a class which is highly coupled tends to be more fault-prone than a class that is loosely coupled. Only one study found CBO to be insignificant while CBO was not captured in 5 studies. RFC was also reported to have a strong significant relationship with class fault-proneness in 24 studies. These findings confirmed that a class with higher response sets tends to be more fault-prone than others with less response sets. Interestingly, none of the studies that measured RFC found it to be insignificant while 5 of the studies did not measure RFC.
3. ***Cohesion Measures:*** In the context of cohesion measures, analysis also shows that 14 studies confirmed LCOM to have a significant impact on class fault-proneness. 4 studies found LCOM to be insignificant while 11 studies did not measure it. The overall results showed that a class with a low cohesion value is more likely to have faults than a class with a high cohesion value.
4. ***Inheritance Measures:*** For metrics that capture class inheritance, analysis shows that only 9 studies found DIT to have a significant influence on class fault-proneness, though with either strong or weak significance. In other studies, analysis indicates 15 studies found DIT to be insignificantly related to fault-proneness while 5 studies did not measure it at all. The insignificance of DIT indicates that a class with a higher number of inheritance depths is not likely to have faults. For NOC, only 3 studies found it to be significant, while 15 studies reported it to be insignificant. NOC insignificance means that a class that is known to have a higher number of children is not likely to be as fault-prone as others with fewer children. Also, 11 studies did not measure NOC at all.
5. ***Class Size Measure:*** Lastly, for size measures that account for all the non-empty and non-commented lines of the body of the class and all of its methods, analysis shows that LOC of a class has a strong relationship with fault-proneness. Over 17 studies confirmed its significance, but no study found it to be insignificant while 12 studies did not consider LOC in their work. The results indicate that a class having a larger number of lines of code is more likely to have faults than classes with small code lines.

6.4.2 Results Discussion

In general, as shown in Figure 6.3, LOC, CBO, RFC, and WMC were the metrics that were most often reported as having strong significant relationships with fault-proneness followed by LCOM. This is in line with the findings in [123][124]. The results were measured as a function of the value of each metric. In this case, *the higher the value of the metrics, the higher the fault-proneness of the class*. In the analysis, DIT and NOC have the highest numbers of insignificant relationships in all the studies. Based on the results, it is believed that the best predictor of fault-proneness seems to vary according to the type of applications used, the language used, and the application domain. Therefore, the recommendation is that all these measures should be considered as predictors of fault-proneness since it is a function of their value in that particular software product. Although DIT and NOC appear not to be regular class fault-proneness indicators, however their significance or insignificance could be as a result of either the developers' experience or the inheritance strategy applied.

In addition, we also found that in order to ascertain the validity of the techniques and metrics used, several empirical studies were conducted in both academic and non-academia environments. In this case, analysis shows that about 79% of the studies were conducted in non-academia environments using systems written mainly by software professionals (using data sets from public repository such as eclipse, NASA, and so on), while 21% of studies were performed in the academic environments with applications written mainly by students. (See Figure 6.3a) Across the projects in these studies, 20% were student's projects, 33% were open source software project (OSS) while 47% are non-OSS systems [18]. (See Figure 6.3b)

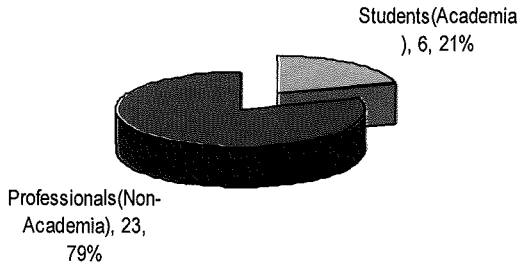


Figure 6.3a: Metric Validation Environments

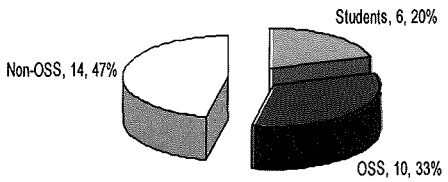


Figure 6.3b: Metric Validation Projects

For the software applications used in all the studies, it was found that only applications written with java or C++ programming languages have so far been used to validate the impact of software measures on fault-proneness of OO classes. This shows that applications written with these two OO languages are dominant in today's software applications. From the analysis, we found that about 54% of the applications were written in C++ in both industry and academia, while 43%

were written in Java and 3% of the studies did not mentioned the language of their application [18]. (See Fig.6.4)

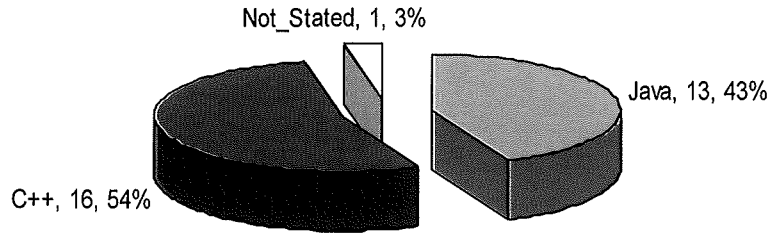


Figure 6.4: OO Programming Languages Used

It is on this note that we have chosen to work in providing maintenance support to CIA of software written with java.

6.5 Process Measures

It is important to know that the reliability of software does not only depend on product characteristics but also on the process characteristics associated with its development. To this end, today process measures are gaining momentum and are becoming essential elements used for more accurate fault prediction models. These measures are computed based on the change properties and faults history of the software. Thus, the process-related measures considered in this study reflect the changes made to a software code over time and the number of faults reported over a period of time maintained in a consistent way in change management systems. In the context of this research, assessing the impact and risk of a change requires a careful understanding of the scope of changes using historical information. Like the OO design metrics, significant efforts have also been directed to creating, using and validating different process measures using real-world software systems [107][115]. In general, these studies have shown that process measures based on change properties and fault history are better indicators of fault than product metrics of code, especially *after-release faults* [107]. Measures related to processes include changes and fault corrections, developer information and experience, purpose and time of changes, file's age, and so on. But *which of the process measures are actually good predictors?*

In an attempt to answer the question, a CLR was conducted on process metrics that are having influences on fault-proneness in several published articles. With the CLR, we chose suitable process measures to be used in building the fault prediction model after consulting previous studies. Table 6.4 captures the different studies, the measures and prediction techniques used. As shown, "++" indicates that the process measures are good predictors of fault-proneness while "--" are poor indicators of faults.

6.5.1 Selected Process Measures

In this section, we give a brief description of the various process measures considered important and are useful for building a good fault model in this study. These measures are grouped into four main groups: *size, people, purpose, and file* or *class.*

1) ***Size Measures:*** Size measures we considered are measures that are based on code churn or deltas committed in the entire history of class. Code churn as a process measure constitute the amount of code change taking place within a software unit over time [115]. This measure can easily be extracted from a system's change history either manually or automatically accounted for by a version control system (CVS). Previous studies have empirically shown that code churn is a good indicator or predictor of fault-proneness of a class. The different size measures are: LOC added, deleted or modified. On the other hand, churn is the sum of all three counts. Table 6.4 shows different studies that have investigated the impact of change size on fault-proneness [107][111][112][113][115]. We have used these studies to support this research with these studies and aligned it to the fact that the lager the change to be implemented, the more likely it would fail.

2) ***Purpose Measures:*** Due to the inevitability of software changes, changes can be performed for different purposes: *corrective*, *preventive* and *adaptive* [3]. In our context, only changes that are corrective, i.e. fault fixes will be the focal point. Corrective change often involves the fixing of reported faults while others could be the addition of new functionalities and sources codes. Previous studies have shown that a change which involves bug fixing tends to introduce more faults than changes that add new functions. A module or class with previously fixed faults is likely to have a fault when changes are performed on them. This could be a function of the developer's experience. Several studies have shown that bug fixing change is a good predictor of failure-proneness of a class [107][108][113]. Furthermore, the number of reported faults linked to a change is also a good predictor of failure-proneness of a class [108][114][115][122].

3) ***Class Measures:*** File or class measure are process metrics which include measures such as the number of files revised or changed, number of fixes in a file (past faults), the change diffusion and age of a file. These measures, for instance the number of past faults, have been found to be good indicators of future faults in a class. Previous studies have shown that if a class has undergone several changes in the past, a change that modifies this class in the present release may be more risky to perform. Several studies have investigated these measures and recommendations given in terms of their predictive capacity. Another measure we found suitable is the age of a class. This measure can also help to predict the

fault-proneness of a class. All the class measures are collected from the change history [108][115][119-122].

Table: 6.4 Reviews of Process Measures

REF.	Measures			Study goal, Statistical Technique	Recommendation	
	Delta	Developer	Past faults		Positive	Negative
[107]	X	X	X	Prediction, C4.5, NN, LR	++ (process measures)	-- (Product measures)
[108]	X	0	X	Prediction, LR, Naïve Bayes, Decision trees	++ (Max/Ave_ChangeSet, No.Revisons, and BugFixes)	==
[109]	X	X	X	Prediction, LR	++ (No. of lines added, no. of bug reports and developer experience)	==
[110]	X	X	X	Effect of developer info, LR, classification tree	++ (Developer Info)	==
[111]	X	X	0	Prediction, Stepwise Regression	++(change bust)	==
[112]	X	X	X	Prediction, LR	++(lines added, deleted, modified,churn)	==
[113]	X	X	X	Prediction Naive Bayes Bayesian networks, LR, Bayesian LR.	++(Bugfixes, No. of Revisions, File Age)	==
[114]	X	0	0	Prediction, LR	++(developer experience and skill)	==
[115]	X	X	X	Prediction, GLR	++ (numbers of changes, File Age)	-- (No. of Developers)
[116]	X	0	0	Prediction, Statistical regression, Spearman rank correlation	++(Relative code churn)	-- (absolute code churn)
[117]	X	0	X	Prediction, negative binomial regression	++(File size, Programming language)	==
[118]	X	X	X	Prediction, LR	++(change diffusion, developer experience)	==
[119]	X	X	X	Prediction, negative binomial regression model	++(Developer info, file size)	==
[120]	X	X	X	Correlation, Spearman Correlation	++ (No. of changes, the no. distinct developers, the file's age)	==
[121]	X	0	0	Prediction & Correlation, negative binomial regression model	==	-- (LOC)
[122]	X	X	0	Prediction, negative binomial regression	++ (No. of Prior changes)	==

** GLR- generalized linear model, **LR-Logistic regression, **x-study measure, **++ - good predictors of faults, ** -- poor predictors, **== - no recommendation

4) ***Developers Measures:*** People measures are process metrics that are associated with the developers who developed or modified the product. Measures we considered in this regard are the number of developers who modified a file and their experience. Previous studies have confirmed that developers' measures are good predictors of faults in a system. Studies have shown that a class modified by many developers would have more faults than a class modified by a single developer, since it involves a mixture of different coding styles and thoughts [107][110][119-122]. Accordingly, inexperienced developers tend to insert more faults than expert developers. However, there are several criticisms that point to the measures of developer's experience as it has been linked to human measurements [110].

Table: 6.5 Summaries of Process Measures and their Description

Group	Abb.	Description	Logic
SIZE	LA	Number of lines added in a change	Changes involving addition of more lines or new features tends to be more risky with less testing
	LD	Number of lines deleted in a change	Changes that delete more lines in file tend to be more risky especially if it remove large amount of codes or incorrect removal.
	LM	Number of lines modified in a change	Change that modified more code lines is like to make wrong changes and is risky.
	CN	Code Churn, sum of lines added, deleted and modified within a change	Changes resulting in large amount of churn are difficult to commit and are therefore risky.
PURPOSE	BF	BugFixes. It is an indication that a change was a bug fix	Changes that fix a fault are said to be complex and are likely to be faulty.
	FR	No. of Faults Reported that are linked to a change	Changes that have been linked to several reported faults required to perform more changes and are risky.
CLASS	CDff	ChangeDiffusion, the number of classes modified by a change	Changes that affect more classes need a good understanding of dependences within different files and are more risky.
	NR	No. of classes revisions or changes are the past changes to the file modified by a change.	Classes that are frequently changed are difficult to modify since their structures tends to degrade. Therefore, a change on such file can be risky.
	NFF	Number of time a class was involved in fault fixes	Classes that are frequently changed are faulty. Therefore, a change on such file can be risky.
	AGE	Age of a class in weeks, counting backwards from the release date and going back to its first appearance in the code repository.	Classes that are old are more likely to be fault-prone when changes are performed on them than new classes.
DEVELOPER	ND	Number of Developers that modified a file	Files changed by several developers are difficult to modify and are likely to be risky with more faults.
	DE	Developer's Experience. It is a count of the number of previous changes done by the developer.	Changes done by novice's developers are more risky than expert developers.

***Abb – Abbreviation*

6.5.2 Strengths and Weaknesses of the SLR and CLR

In the SLR and CLR we conducted in this research, we covered a large number of articles that assisted in extracting the relevant information we used. At this point, we are confident that the study actually covers the empirical validation of CK + SLOC metrics, process-related measures and the models that utilized them. We strictly followed the guidelines by Kitchenham et al [106] and the steps to perform a good CLR using credible and trusted sources considered to be rich and recognized by research bodies worldwide.

However, possible threats to the validity of the findings could be due to the search terms used, the risks posed by not covering all the relevant studies, or some relevant studies could possibly be hidden in the sources excluded, as well as the risk of misrepresenting the findings of some of the

papers found, such as not considering results obtained at different fault severity levels, as well as the level of significance and insignificance of the metrics. But we are very confident that if a few relevant studies were not found, the information they contain will have no significant effect on the results presented. To counter these, we worked collaboratively and analyzed all selected studies, all decisions and results were checked, rechecked and all inconsistencies resolved.

6.6 Study Theoretical Hypothesis

The theoretical basis for choosing the different *product* and *process* measures to be used in building quality model in this thesis is stated in this section. It is based on the theoretical basis that underlies the development of quantitative models that relate OO metrics, process metrics and fault-proneness [11][41][103]. This thesis therefore hypothesized that, the relationship is due to their effect on cognitive and structural complexity. This is captured in Figure 6.5 and the indication is as follows:

1. The structural properties of a class (e.g. coupling, cohesion, inheritance) impact cognitive complexity which in turn, relates to the fault-proneness of the class. Accordingly, high cognitive complexity will lead to a component exhibiting unwanted external qualities like fault-proneness, reduced understandability and maintainability (e.g. CIA). Metrics that have the ability to measure these structural properties would be considered good predictors of fault-proneness.

2. Process properties such as change and fault properties will affect or increase the structural complexity of the software which is related to fault-proneness. As changes are made to software, its quality tends to degrade and the structure becomes very complex to maintain. Thus, changes related to sizes, frequency of change, fixes, experiences, and so on could introduce faults to the software class. This means that metrics that measure these change and fault properties would be considered good predictors of fault-proneness.

In general, fault-proneness of OO program classes can be impacted by:

- the structural features of classes,
- the volume of change or fault fixes carried out by the class in the past,
- the quality of the of class coding, and
- the expertise of each individual developer performing the changes.

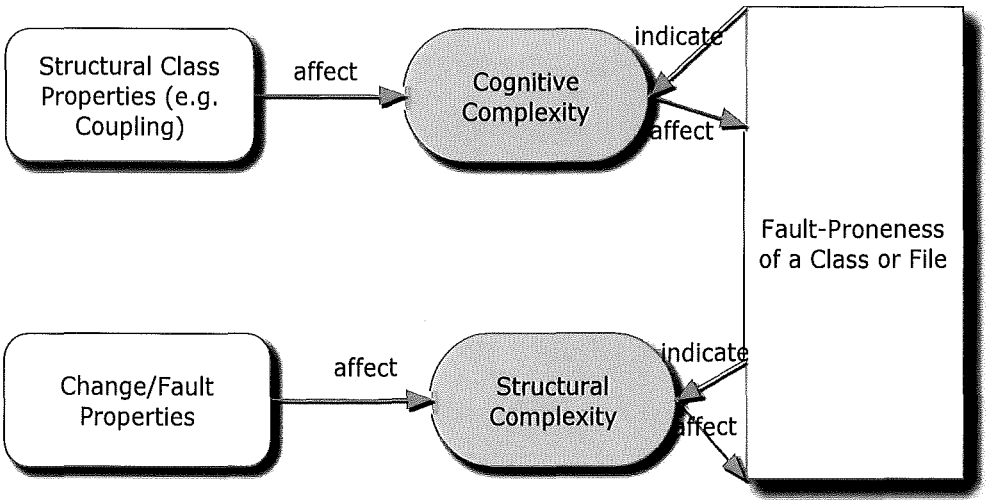


Figure 6.5 Theoretical Bases of Process and OO Product Metrics

The above expression can serve dual vital purposes: the early prediction or identification of software components that are of high risk and the construction of preventative (e.g. design, programming) strategies. In this case, Size, CK metrics and change/fault history would help in assessing the risk of a change during CIA at a reduced cost, in order to take solution actions early and thus avoid costly rework.

6.7 Change Data Managements

6.7.1 Development Activities

In order to have an in-depth understanding of how process measures are maintained, this section provides a development model that this research assumed. Since changes have become a common routine that is inevitable, during the course of development, software components are thus created or updated by either one person or multiple persons. In order to account for what has been done with respect to *how*, *when*, *where*, *why* and *who*, specific software change repository with CVS is indispensable. More than one person is allowed access to a class or package to perform addition, deletion or modification at the same time. As changes are implemented by a developer, all details are captured and stored in the CVS. The software application is then set to undergo a series of changes known as *revisions*. After completing revision iteration, it is then released with a version number and a new revision begins for a new version. Usually, sets of related changes are grouped together in a *release, **R**, a* practice that is consider good and aimed at ensuring customer's satisfaction. However, due to the increase in size and complexity of software applications, releases are often implemented as software updates.

To make changes in software requires the use of an initial maintenance request (IMReq) [118]. It is a change that either corrects faults, perfects or enhances some part of the system or introduces new features. A feature is implemented by a set of IMReq. For effective management, each IMReq is structured into sets of maintenance requests (MReq). Each of these MReq can be restricted to a distinct subsystem which represents a solution that needs to be solved by a problem stated in IMReq. A single IMReq can implement one or more MReq. For operational accountability, each developer owns one MReq and it represents all or part of the developer's role in the solution to an IMReq. Changes to several source code classes may be included in one MReq. As classes undergo several changes several times, each atomic change is recorded as a *delta,* sometimes called a *churn* which is stored by a CVS. Regardless of whether a change is implemented in a small or big release, it carries with it the probability of failure. We deem it important to identify early which modules or classes affected by an IMReq will have a high probability of failure in the field. This then forms the motivation for this research - the probability of failure for IMReq. In subsequent sections, we shall discuss how change data can be managed in a CVS.

6.7.2 Change Data Repository

In previous studies, the importance of change data in relation to accurate and better fault prediction has been emphasized. However, process measures in this regard have been ranked as the most difficult to extract. Some of the issues that might account for this problem include:

1) Change repositories are usually not structured in a way that supports the mining of required information for fault prediction purposes.
2) On a regular basis, it is cumbersome to decide which change reports in a database represent faults.
3) Lastly, developers and testers in a software project may record information that is not accurate into the IMReq database.

Particularly, in the study by Nagappan et al. [111], it was stressed that one of the problems is due to how the repositories housing the change and fault history are structured. For instance, CVS and BUGZILLA have been the most widely used change data and fault repositories respectively [125]. Several studies have shown that BUGZILLA lacks some basic information such as *purpose of a change*, e.g. did it fix a bug? It is this change purpose that facilitates the identification of different fault reports such as mapping bug to fixes and the locations in the code that caused the problem. Thus, in order to effectively collect the needed data, the repositories have to adapt to the development process followed by such organization. This thesis therefore describes an effective approach that will assist a maintainer in capturing the change data required in order to associate

these measures with *before* or *after-release* faults. Based on the structure of the existing CVS and bug tracking systems like BUGZILLA, for the analysis in this study, we designed change and fault repositories - IMReq and *delta* database.

6.7.2.1 IMReq Repository

This is the change database that stores information for a developer who commits change about an IMReq. The basic information associated with IMReq database includes:

1) *The date the request was made,*
2) *The date the last delta of the IMReq was completed,*
3) *The affected files,*
4) *The developer who originated the IMReq,*
5) *The module changed per IMReq, and*
6) *The purpose of the change (fix or new).*

Among the IMReq database list, the *purpose of a change* is the most critical field. Therefore, a good classification of the nature of faults will make a lot of difference in facilitating their identification. In previous studies, *purposes of a change* in MReq were marked with keywords such as "fix," "bug," "error," and "fail" to represent changes classified as corrective. Also, bug status in BUGZILLA are UNCONFIRMED, NEW, ASSIGNED, RESOLVED, or CLOSED and the RESOLUTION, e.g., FIXED, DUPLICATE, or INVALID [111]. Therefore, we mirrored these databases to represent fault reports so as to speed up the correct identification of corrective faults in the IMReq database.

6.7.2.2 Delta Repository

Delta repository (DR) is more or less change management databases where tracks of deltas involved in each IMReq are maintained [118]. The various attributes recorded as delta in the delta repository are as follows:

1) *Change date,*
2) *Names of the class or module changed,*
3) *Number of line added, deleted, and unchanged,*
4) *ID of the developer who commit the change,*
5) *IMReq identifier,*
6) *Release identifier.*

The information stored in this repository is known as the change history, which is critical to the building of our failure probability model as independent variables, in addition to the product measures. In previous studies, all the information about each MReq is stored in an Extended

Change management system (ECMS) database while the Source Code Control System (SCCS) is used to maintain all information in the DR [118].

6.7.3 Change Data Organization

To organize the change management system for speedy and correct extraction of the essential change and fault data, the following requirements are important:

1. Firstly, we have to ensure that the source code is organized into package and each packages subdivided in classes which are made up of a number of source files.
2. Secondly, we need to ensure that each change has to be recorded as an IMReq using an IMReq tracking system (IMReqTS). The IMReqTS will keep tracks of ***R***, IMReq and fix concerning faults found during testing or in the field.
3. Lastly, Release Failure Database (RFD) has to be maintained to keep track of all reported faults or failures associated with the release and the IMReq that caused it. In the context of this study, the importance of the RFD is to help account for all failures associated with a release after changes have been made and the IMReq that is involved. (see Figure 6.6)

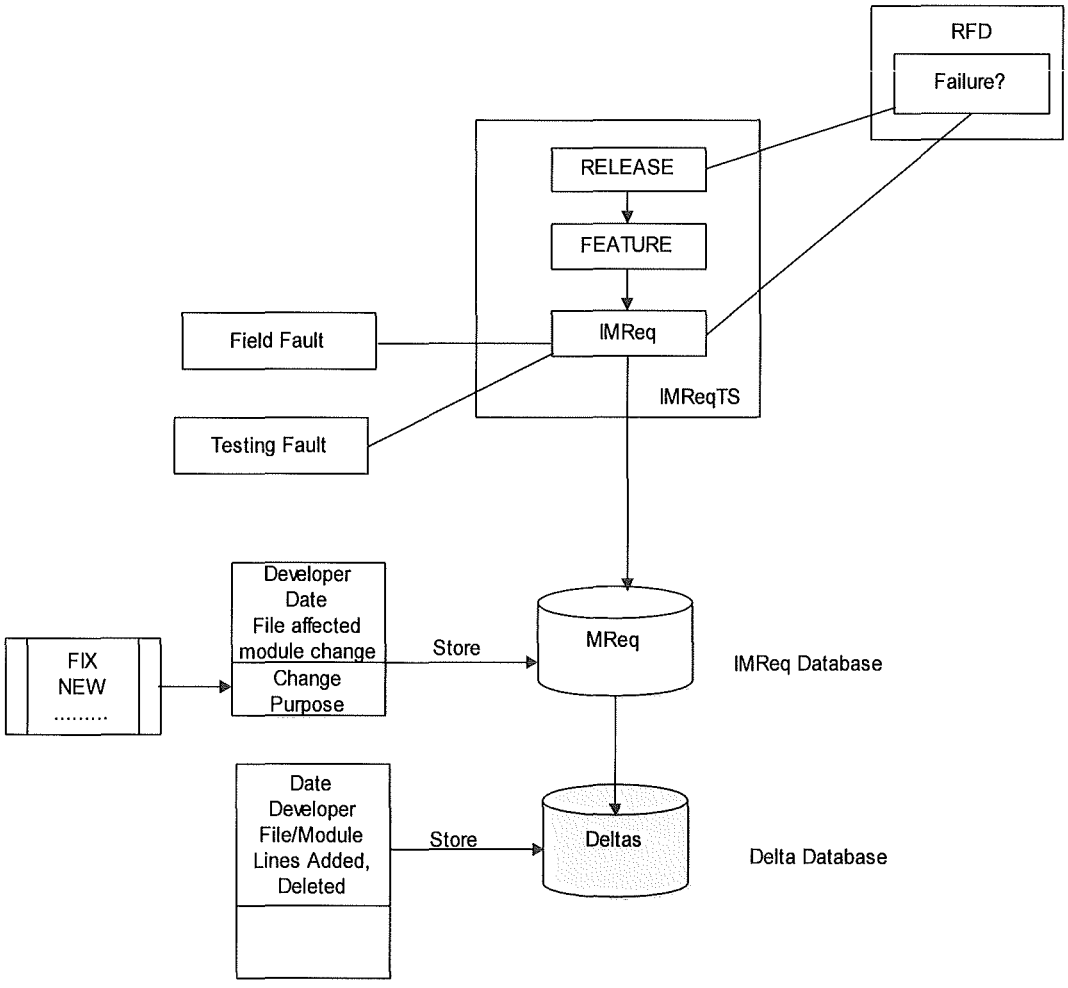


Figure 6.6: Change Data Management

In Figure 6.6, the change of a class about a particular MReq is maintained using IMReq database and within each MReq, every change made to a class has to be recorded as a *delta* in the delta repository (DR). Albeit all MReq changes restricted to one file can be implemented by one delta, for changes considered to be large, it is not always so in practice. In this case, developers can carry out several deltas on one file. In general, the measures or data used in this study are extracted from both IMReq repository and DR which includes the lines added or deleted, the purpose for the changes, and so on. We adopted these data because they reflect the different ***chtype*** involved in our CIA technique which can easily be computed from the change management system.

6.8 Measuring Modification

In this section, we give a brief description of how to measure the degree or volume of changes in a particular ***R*** and the number of developers involved in the change as shown in the **IMReqTS** of Figure 6.6. The basis is to assist a maintainer in knowing which added or removed information can substantially contribute to the prediction of fault-proneness of a class. In this research, we measure the number of changes in previous **R-n**. For several releases R_i, where $(i=1,2,3,3....n)$, we consider only **R-1** and **R-2** as the two key releases that will form part of the model as training data to be used to predict the fault-proneness of classes in release, ***R+1***– future release. We therefore discuss this information as follows:

6.8.1 Add, Delete and Modify

During the course of performing changes, three types of changes are important: numbers of lines added, deleted, or modified in the class. To understand what was performed in each change or its volume each time a developer performs an update on the version of the file, the following definitions applies. As shown in the DR of Figure 6.6, each change made is a function of the sets of lines in the class. For example, if a developer changed three different lines in a program separated by at least one unchanged line, we considered it as three different actions that occurred separately.

i. If the action that applies is either only to add new lines or only delete existing lines, we referred to this as either the *number of lines added* or *deleted* respectively.
ii. If the applied action is modifying each line without adding or deleting other lines, we referred to it as *the number of lines modified.*
iii. If the applied action replaces λ connecting lines with λ + i lines (where $\lambda>0$, $i>1$), we referred to it as two separate counts: λ *lines modified* and *i lines added.*
iv. In the same order, if the action replaces λ lines with λ − i lines, we thus, count it as: $\lambda - i$ *lines modified* and *i lines deleted* respectively.

v. The sums of all respective add, *delete*, and *modify* actions done to a class in that ***R*** is called the *churn* counts.

In this research, we believe that measuring changes in this direction is a good step toward accurately accounting for all volumes of changes in a particular version in a ***R***. Normally, if we only define changes with respect to *add* and *delete*, an action that replaces λ lines with λ+i could be counted as deleting λ line and adding λ+**i** lines. Nevertheless, it is of the essence to separate these actions because, though MReq changes restricted to one class can be implemented by one delta, developers can perform multiple changes that would involve multiple actions in a class in a single ***R***.

6.8.2 Number of Developers

Another aspect of change measurement is the count of developers that actually modified a module in earlier releases, ***R-1*** or ***R-2***. Even if one MReq is owned by a single developer, a module can be modified by several developers. In this case, a measure of the number of distinct developers that change a class in its history could be valuable to prediction. The fact remains that classes that were changed by many developers are more likely to be fault-prone than modules modified by a single person. Thus, in this study we provide information on how to count the number of developers who changed a class and the number of classes a developer has changed which can be used as a surrogate measure of developer experience.

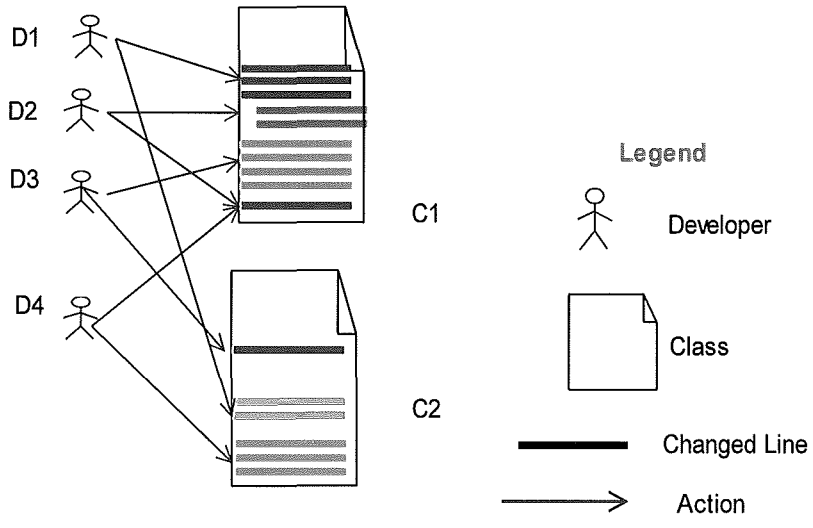


Figure 6.7: Developer's Change Activities

Let C_i (i= 1,2,3…,k) represent the classes in a program that has undergone changes in the IMReq in ***R-n***. Let D_k (k=1, 2, 3,…, n) be developers who committed the changes on the C_i. If we

consider two classes: C_1, C_2 and four developers' D_k activities on them, the following measures can be obtained:

1) No. of classes modified by a distinct developer, (NCM)
2) No. of commitments per developer in a release,(NC)
3) No. of lines committed by a developer in the release, (NLR)
4) No. of Developers that changed a class,(ND) and
5) No. of lines of code changed by each developer in a single class (LCCD).

Table 6.6: Developers and Class Information

Dev.	Measure				Class	Measure				
	NC	NLR	NCM			ND	LCCD			
							D1	D2	D3	D4
D1	2	5	2							
D2	2	3	1		C_1	4	3	3	4	1
D3	2	5	2		C_2	3	2	0	1	3
D4	2	4	2							

In Table 6.6, developers' activities in Figure 6.7 in relation to the classes they modified are presented. The first part of the table contains the developer's information. Here, for instance, information for D1 shows that, during the course of development, D1 made 2 separate sets of changes, changed 5 lines of code in the modules, and have modified 2 different modules. In the same vein, in the history of C_1, 4 different developers have worked on it, where D1 and D2 have changed 3 lines, D3 changed 4 lines and D4 changed only one line. The benefits of these counts stems from the fact that knowing developers' activities and the faults associated with them would assist in predicting the fault-proneness of a class or module.

In this research, the number of distinct developers (ND) that modified a particular release will be used. Accordingly, a developer's experience (EXP) will be computed using surrogate measures which are based on previous modifications done by the developer that have been linked to fault reports. In addition, other measures of interest will be computed as well. For instance, size of the class will be computed in thousands lines of code (KLOC) and the class's age will be computed based on the number of past releases the class was part of the system.

6.9 Chapter Summary

In this chapter, we have presented the different product and process metrics that have been empirically validated as having an impact on class fault-proneness using real-world software in industries and academic environments. These metrics will therefore be used in the chapter that follows to construct the fault prediction model. In addition, the management of process metrics in a software project was also discussed.

CHAPTER 7

Early Fault Prediction

7.1 Overview

This chapter is centered on the construction of a software quality model to support CIA activity of large-scale OO program maintenance. It starts with the chapter introduction, faults and failure relationship and then advances to the discussion of the processes involved in data collection, model design technique and model evaluation. The aim is to build a fault prediction model that will assist developers to determine the probability of failure of a component affected by a change before a change is made.

7.2 Background Information

In today's e-world, software is becoming extremely critical and pervasive due to increases in software dependability. For software to satisfy the quality attributes of dependability, its size and complexity has to increase in order to handle all the aspects considered critical. The development of these large systems can span through many years and go through series of evolution which tends to degrade their structures and make them more cumbersome to understand and change further. When changes are difficult, they become costly and take a longer time to complete. Thus, difficult changes could introduce excessive numbers of faults during the process of changing the code due to hidden faults in the systems' design, code, or the change process itself. In some cases, the system could become unstable or uncontrollable, and this increases the risk of field failure or results in failure.

In the field of Software Engineering, fault prediction is an important activity that has been used to assess the quality and complexity, or to estimate if the quality standards enforced by the customer satisfaction are actually realized in the final product. Several researchers and practitioners have studied a number of fault prediction models and the fault-proneness of classes have been identified with a high degree of accuracies prior to testing by utilizing these models [11][45][80][87][90]. These models have been very useful in detecting the number of faults in a software system. This is because it is believed that the software testing activities and code inspection consume a substantial amount of resources such as money, time, and staff, especially on fixing faults discovered after software has been released to the users [105]. In addition, recent evidence has shown that the probability of fault detection accounts for 71% of faults using prediction models is higher than that of software review which account for 60% of fault detection before testing [123]. Thus, it is cost-effective to use fault prediction to detect software faults.

The studies we considered in the CLR and SLR which have investigated the relationship between software metrics and fault-proneness, were conducted in different contexts and differ from this research since prediction was either employed to assess the quality of software products (in design or code) of a particular release before testing [80][81][95][107][109], and were not employed in CIA process. The only study that is almost similar to this research is the study by [118], where the risk of failure probability was measured after a change had been implemented, but still not during CIA. Therefore, in this thesis we aim to build a software fault prediction model that will be used to support CIA activity in order to reduce or eliminate the risk of software failure by focusing verification and validation activities only on components that we considered fault-prone and are affected by a proposed change using the structural properties of the OO program, change history and fault corrections history of a particular ***R***.

7.2.1 Early Fault-Prediction Benefits

It is believed that when software faults are predicted early on classes, it will help software testers to focus their efforts on those high risk classes, allowing them to identify faults more speedily and take necessary action. Consequently, it will yield a system that is of higher quality, with fewer faults, stay within the project budget and deliver the product on or before schedule. In the context of this thesis, the reasons for adopting fault or failure prediction during the course of CIA in OO programs centered on the following benefits:

1) Due to the criticality of today's software, producing a system that is highly dependable is important.
2) Since changes are inevitable, and dependency analysis doesn't guarantee the absence of faults after a change has been implemented, predicting faults early will help to improve verification and validation activities by targeting fault-prone components.
3) To validate and select best measures of OO metrics and change activities that have impact on the fault-proneness of classes, and
4) To improve the overall quality of the software by improving testing activities.

These benefits form the motivations behind this study, and the section that follows, explains the differences between faults and failure.

7.3 Fault and Failure Relationship

In this section, before explaining the data extraction process, we quickly delve into exploring the difference between fault and failure in the context of this thesis.

i. ***Fault:*** Fault is an error in the software that triggers it to fail in carrying out its required function. It is a hidden error that may or may not actually manifest as a failure in the software system.

ii. ***Failure:*** A failure is an observable or a manifestation of an error in the program behavior. With failure, the software behavior deviates from it specified behavior, and does not do what the user expects.

With these definitions, though there is a slight difference between the two, every failure can be mapped back to some faults, but not all faults result in a failure [124]. However, in any case, the consequences are enormous and costly. In the context of this thesis, faults and failure will be used interchangeably in the prediction of failure-proneness of OO classes during CIA. We achieve this by predicting the fault-proneness of classes and generalize it to failure based on the probability value using the logistic regression. The rationale is that, it is only faults that exist in software classes and if left undiscovered they may result in software failure. Thus, it is this failure that needs to be avoided by identifying the faults early during the development.

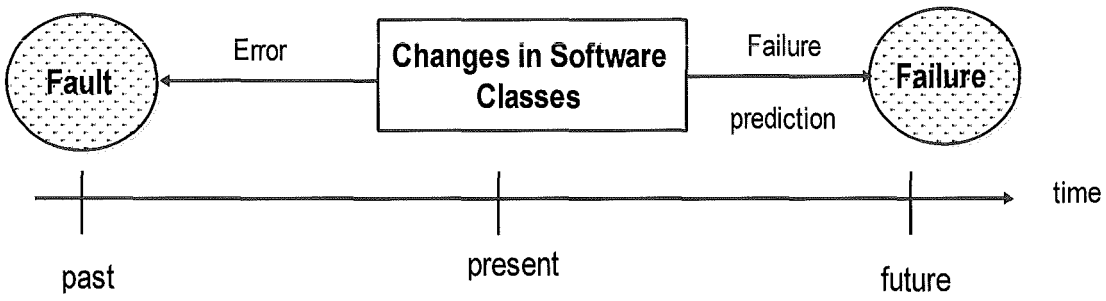


Figure 7.1: Fault and Failure

In Figure 7.1, the relationship between fault and failure in a software system is presented. It shows that, if there is a fault hidden in a class and changes are carried out on the classes which are unconnected with its correction, there is the *probability* that the class will fail. In this case, failure prediction is employed to try to assess the risk that the change will lead to future failure. Thus, to predict fault in classes, product measures (*structural properties of classes* and *size*) and process (*faults and change data*) are used.

7.4 Data Collection Methods

Data collection is a key component of this thesis involving fault-proneness prediction that hinges on information collected from both CIA and software evolution history. The prediction model will principally focus on the classes known to be affected by change request and their code, change and fault measures. In the same vein, fault data will be collected from the bug tracking system,

change data from the CVS, and code metrics (CK + SLOC) will be collected from the snapshot of the source code under study. The processes involved are discussed in subsequent sections.

7.4.1 CIA Data

All the measures to be collected for prediction purposes would be based on the outcome of the impact analysis process. Given a change proposal, the first step is to carry out CIA using the impact prediction model discussed in chapter 5. The output of this activity will be fields, methods, classes and packages identified to be affected by the change. Since classes are the unit of analysis in this prediction model, all affected member fields and methods, except package will be considered in their respective classes. In other words, the resulting classes will be the product of the total impact sets, TIS. After the impact set identification, the next important step is to collect the fault and change histories of those classes to be used for fault-proneness prediction.

7.4.2 Faults Data and Change History Extraction

During the course of development, problems like faults are usually reported by developers or users after the product has been released to the market. When such faults are reported, they are typically recorded in a fault tracking systems such as BUGZILLA, usually over a fixed period [125][126]. In the context of this thesis, the Release Failure Database (RFD) is used. (See Figure 6.6) It is the duty of the developers to locate and fix the reported faults by performing changes on one or several classes associated with the fault. The first step towards extracting fault data is to carefully examine the bug tracking system, RFD in order to identify which components failed and which did not fail when changes were made in *previous releases,*(***R-n***). For the analysis in this thesis, two important kinds of information are investigated: *change data* and the *number of faults/failures* reported in that *release*, ***R***. (Note: *Previous release* (***R-n***) refers to the software system before the proposed change while *next release* (***R+n***) is the software system after the change has been implemented).

In order to achieve this, the number of changes and fault corrections performed on release ***R-1*** that are important to obtain the present release, ***R*** of the software under study are captured. This is necessary because the amount of change for classes in release ***R-1*** is expected to affect the probability of fault fixing in release, ***R***. Data in the archives of the CVS (i.e. IMReq and delta databases) and the RFD in ***R-n*** will be collected. The steps involved are described as follows:

i. Identify reported fault or failures from the failure tracking system, RFD before and after a release.
ii. Classify classes as fault or failure-prone or not by associating faults or failures that involve fixing to the changes in the version where they originated from.
iii. Identify *Before* and *After-Release* faults

In previous researches, *before* and *after-release* faults or failures are called *pre-release* and *post-release* faults or failures respectively [107][125]. Though in previous studies [126], it has been noted that *after-release* faults or failures are more important than *before-release* faults or failures, in this thesis we focus on both. The essence of *after-release* faults or failures in previous studies stemmed from the fact that they are observed and reported by customers. Consequently, they seem to have a higher effect on the quality of the software as alleged by the customer. In the same vein, *before-release* faults or failures are also important as they may be used in predicting *after-release* faults or failure. However, most studies criticized *before-release* faults or failures as not being important since it is a function of testing activities.

7.4.2.1 Fault Identification

Faults in the RFD, are usually documented in the bug reports, with a description of where they originates from: development/testing or the field by users. Both faults and failure are collected and counted separately. In this study, as stated in Section 6.8 of chapter 6, we are only interested in faults that were reported after a given ***R*** and were all fixed. In this case, only faults or bugs with the status **"FIX"** are essential since they are bugs emanating from faults or failures. One practice that is common among developers is associating a bug report number with the comment of a fix fault. This is employed in this study in order to facilitate the mapping of changes to bugs.

7.4.2.2 Linking Before and After-Release Faults to Classes

After identifying all *before-release* and *after-release* faults, the next step is to link the bug reports to their IMReq to identify the changes in the source code that were involved in the fault fixing. In this case, the class where the original bug resides is identified. As stated in the version archives, MReq of every change contains the purpose of such changes which is stored with different status such as NEW, RESOLVED, ENHANCED, and FIX, and so on. Only bug reports with the status "FIX #", "BUG #" will be considered where "#" is the bug number. This is achieved via a manual search to identify all the "FIX" keywords in the version archive. Moreover, knowing the location of every fixed fault found, the number of faults or failure per locations and ***R*** can be counted for.

7.4.2.3 Faulty Class Classification

Knowing the number of classes in the OO software that are faulty and non-faulty would play a critical role in the prediction of fault-proneness in the ***R+1*** when changes are carried out. After locating the origin of "FIX" faults and thus, the classes involved, can then be classified as either faulty or non-faulty using the changes that corrected the *before* or *after-release* faults. A class is classified faulty if it has at least a fixed fault associated with it, otherwise it is considered non-faulty. That is, a component is fault-prone in a release, ***R*** if it was changed in order to fix a *before* or *after-release* fault R. (See Figure 7.2) The outcome of the classification are numbers of classes

with *before* and *after-release* faults. We distinguished *before-release* from *after-release* faults using Zimmermann et al [125][126] approach as follows:

1) *Before-Release faults:* If a class has any *before-release* faults, it will be tagged faulty; otherwise the class is tagged as non-faulty even though it has *after-release* faults or failure.
2) *After-Release faults:* If a class has any *after-release* faults, it will be tagged faulty; otherwise the class is tagged as non-faulty even though it has *before-release* faults or failure.

The general process involved in performing a successful data collection is illustrated in Figure 7.2. In Figure 7.2, for instance, Bug 5555 is reported within a certain period after release ***R-1*** and is associated with that version. After the stipulated period, it wasn't fixed and still exists in ***R***. After ***R,*** the version was associated with the current version and the bug was finally fixed. In this case, bug 5555 have to be counted as either *after-release* fault or failure of **R-1** or *before*-release fault or failure for ***R*** since it was already reported before the release, **R**. Accordingly, the corresponding change and the developer are identified.

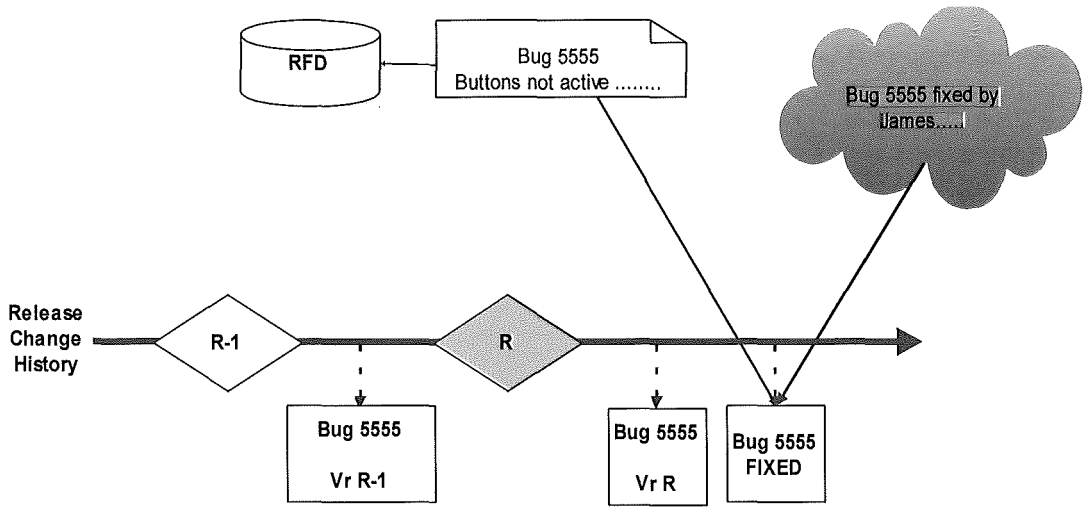


Figure 7.2: Locating and Linking *Before* and *After-release* Faults to Changes

7.5 Prediction Measures

In this thesis, classes are the unit of analysis for identifying all fixed locations in the version archive. Classes are the entities for which fault data will be collected as well as on which failure-proneness prediction will be based. Thus, for each class, the change data and the number of source code metrics (OO M + SLOC) will be computed.

7.5.1 Change Data

File or class level change data for a particular release will be collected via the configuration management system. In the context of this research, a file is equivalent to Java classes. As shown in Figure 6.6, the change data for each class will be collected for each release from the IMReq database using a unique MReqID, which ensures that the change history of a class can be traced even in cases where it changes location in **R**.

Table 7.1 Software Metrics for Fault-proneness Prediction

Factors	Metrics
Process	LA
	LD
	LM
	CN
	FF
	FR
	CDff
	NR
	NFF
	AGE
	ND
	DE
OO	CBO
	RFC
	DIT
	NOC
	LCOM
	WMC
SIZE	SLOC

7.5.2 Object-Oriented and SLOC Metrics

The structural measures of a class will be collected directly from the **R** to be predicted. Unlike change and fault data that are measures from ***R-1***, a structural measure for classes is computed from a snapshot of the codes in the current ***R*** to be predicted due to variation in quality between the two releases. Thus, a third-party metric collection tool called *Analyst4J* will be used to compute all the six CK metric suits and SLOC for each class in the given release. In this thesis, CK metric suit which captures measures of coupling, inheritance and cohesion of an OO program are used. As shown in Table 7.1, 6 CK metric, one size metric, LOC and 12 process metrics of both change and developer information will be used. These all form the independent variables to be used in the perspective of the fault prediction model. In combination with the size, change and fault history, they will be used to predict the fault-proneness of classes.

7.6 Prediction Model

When constructing a fault-proneness prediction model, several decisions have to made about which statistical technique is to be applied, the variables to be used, evaluation method and evaluation criteria [87][107]. However, based on the analysis of the SLR conducted in [18], logistic regression (LR) was found to be the best and most widely used statistical technique used to study the influence of code or change attributes on fault-proneness of classes, where fault-proneness is the most widely used *dependent variable*. This is captured in Figure 7.3a and Figure 7.3b and Table 6.3 of chapter 6. In addition, several model evaluation criteria have been applied to assess the accuracy of the models such as *confusion matrix*, *principal component analysis*, and so on. Thus, in this section, we give a description of the fault prediction model built that will assist maintainers to improve the quality of the software under maintenance, CIA by facilitating testers' work and preventing costly failure in the field. We first identified the variables: *dependent* and *independent* and then proceed to describing the model construction techniques and the evaluation.

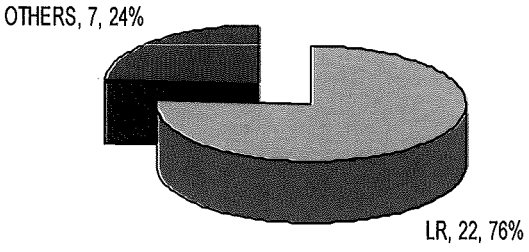


Figure 7.3a: Statistical Techniques Used for Fault Prediction

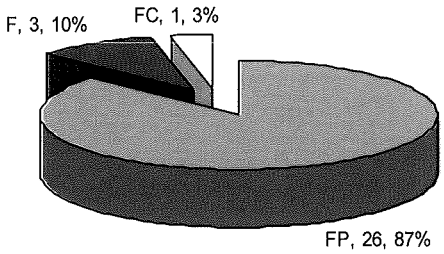


Figure 7.3b: Dependent Variable Used

As presented in Figure 7.3a and 7.3b, F = fault, FP=fault-proneness, FC=fault count and LR= logistic regression.

7.6.1 Model Parameters

In the context of this thesis, the goal is to construct a prediction model that will assist developers' in their CIA activity to detect in advance, which classes affected by a change will be faulty or not. We have to empirically establish the relationship between software measures and class fault-proneness. We use LR to predict the probability of fault-proneness, which requires *independent variables* (i.e. code and change date) being used to predict the *dependent variable* (fault data). Therefore, in this section we identify which measures are to be used and which are significantly related to the fault-proneness.

7.5.1.1 Dependent Variable

The measure of fault-proneness is the suitable dependent variable chosen for analysis in this thesis. We chose this measure because:

1) It is the most widely used dependent variable in the history of fault prediction in relation to code attributes and change history, (see Figure 7.3b)
2) We want to empirically assess the relationship between OO metrics, size, change information and fault-proneness.

Supposing testing was conducted correctly and thoroughly, the probability of detecting a fault in a class due to reported failure by system testers or users in the field should be a good pointer of fault-proneness. Fault-proneness will assume a binary type 0 and 1, where a class is either marked as faulty or not regardless of the fault content in each class. In this thesis, two dependent variables will be used, the count of system test faults or failures for a class (*Before-release* faults) and the number of *After-release* failures for a class in a given release, ***R***. Since the goal is to measure the fault-proneness of a class affected by a change in a specific release, ***R,*** during CIA, where a fault correction could span to many classes, the number of distinct fault fixings that are required in each class for predicting release ***R+1*** when changes are implemented will be counted. These counts of failures or faults are computed from the change history information all together. (See Figure 7.2) In addition, with the uneven fault distribution in OO programs, class fault-proneness in **R** will be treated as a classification problem and is estimated as the probability that a change in a given class causes the software to fail in ***R+1***. To have a clear understanding of the dependent variable used in this study, we give a typical illustration in Figure 7.4.

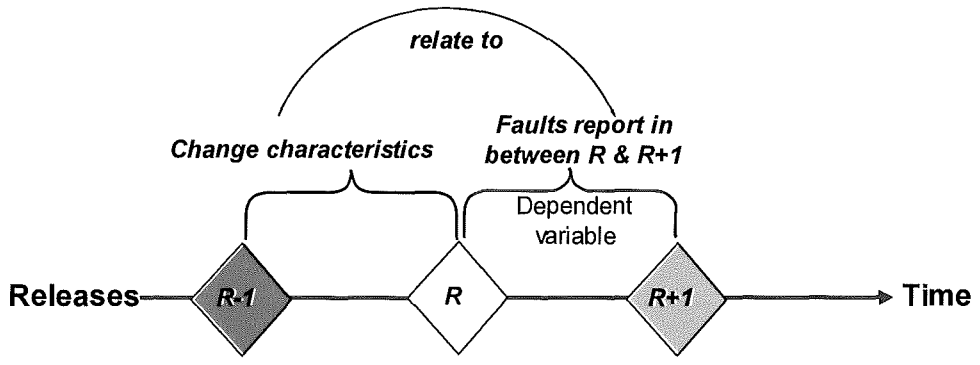


Figure 7.4: Dependent and Independent Variables

Figure 7.4 captured the dependent and independent variables analysis in **R**. It shows the number of faults in the system's classes that occurred between two successive releases, ***R*** and **R+1** during its history either in the field or by developers being used as dependent variable. Thus, the dependent variable is the number of faults in classes reported after release **R** and before release ***R***

+ *1*. To predict which class will be fault-prone in ***R+1***, changes carried out in ***R*** or ***R-1*** of a class are related to the number of faults reported for that class in release ***R*** and ***R+1***.

7.6.1.2 Independent Variables

In the context of this research, software measures of size, structural properties of OO classes and change history are the independent variables. These measures are captured in Table 7.1. The suitability of these metrics stems from the fact that they have been empirically validated and their relevance in building quality models in industrial software development have also been confirmed [93]. In this state of affairs, many studies have shown that the structural properties of OO classes are connected to fault-proneness. (see Table 6.3) In addition, we also considered size measure, LOC which has also been confirmed in previous studies as having a strong relationship with class fault-proneness. For the *process measures*, several studies in the literature have proved their usefulness as good predictors of fault-proneness [107]. These measures are shown in Table 7.1.

7.7 Prediction of Before and After-release Faults

In this study, two different *independent measures*: codes and change characteristics will be used to build the prediction model but in different contexts. We will propose measures for predicting *before-release* faults and *after-release* faults respectively. This is discussed as follows:

1) *Before-release* fault prediction: When building a prediction model for predicting *before-release* fault-proneness of classes, we recommend the use of size and OO design metrics, CK metric suit in particular. The choice of metrics is based on analysis and recommendations of several empirical studies that have been carried out. These studies have proved that CK+SLOC metrics proffer ways to evaluate the quality of software and using them in the earlier phases of software development can assist firms in assessing large software development fast and at a low cost [11][40]. Hence, using these metrics to predict *before-release* fault-proneness will help testers to improve their effectiveness and efficiency.

2) *After-release* fault prediction: In the same vein, when building a prediction model for predicting *after-release* fault-proneness of classes, we recommend the use of size and change history measures. The rationale is that, in a study by Shatnawi et al.[87], OO design measures were considered as poor predictors of fault-proneness of *after-release* products. It was found that OO design measures cannot be used to build a model to identify fault-prone classes with satisfactory accuracy as the system evolves. Thus, alternative metrics should be used if high accuracy is to be achieved. Another study by Arisholm et al. [107] found that using OO design metrics measures to construct fault-

proneness prediction models will not yield cost-effective models. The reason is that OO metrics correlate strongly with size related measures. Therefore, in this thesis we believe that using measures related to change properties themselves will be a good indicator of fault-proneness in after-release products. Previous changes and fault data could be valuable to help in fault-proneness prediction via the identification of classes that have been indicated to be naturally faulty and always affected by changes in the past.

In both cases, the knowledge about certain features of software that are indicators for fault-proneness is very valuable especially for software testers due to the fact that it will assist them to focus the testing effort and to allocate their scarce resources correctly.

7.8 Model Construction Techniques

Software fault prediction models have been known to play a critical role in software quality assurance as classes which are likely to contain faults are identified early during CIA. This section provides a detailed description of the techniques used to build quality prediction models. In this case, the quality of each class using fault-proneness data available from the same or similar types of previous projects and software measurements are used as training data. LR is used to model the connections between the measurable structural properties of a software product, change characteristics and their fault-proneness. (See Figure 7.5) To construct this model the following steps are followed:

i. Identifying model parameters – dependent and independent variable (see section 7.5),
ii. Mining of the metrics for significance relationships,
iii. Formulating an LR model that measures the independence relationships between the variables of interest, and
iv. The resultant distributions and description of the model evaluation criteria.

Once this model has been constructed, the next step is to validate it empirically. (See Figure 7.5)

Figure 7.5 shows how software measures can be used to construct a fault prediction model. The hypothesis underlying the model is based on the practical belief that these measures will capture the quality of the software product. By discovering the information stored in historical project data, structured in the form of software measures and fault data, an effective software quality prediction model can be built. Fault-proneness is dependent on a set of variables which are independent of the programming language used, but simple and freely-accessible.

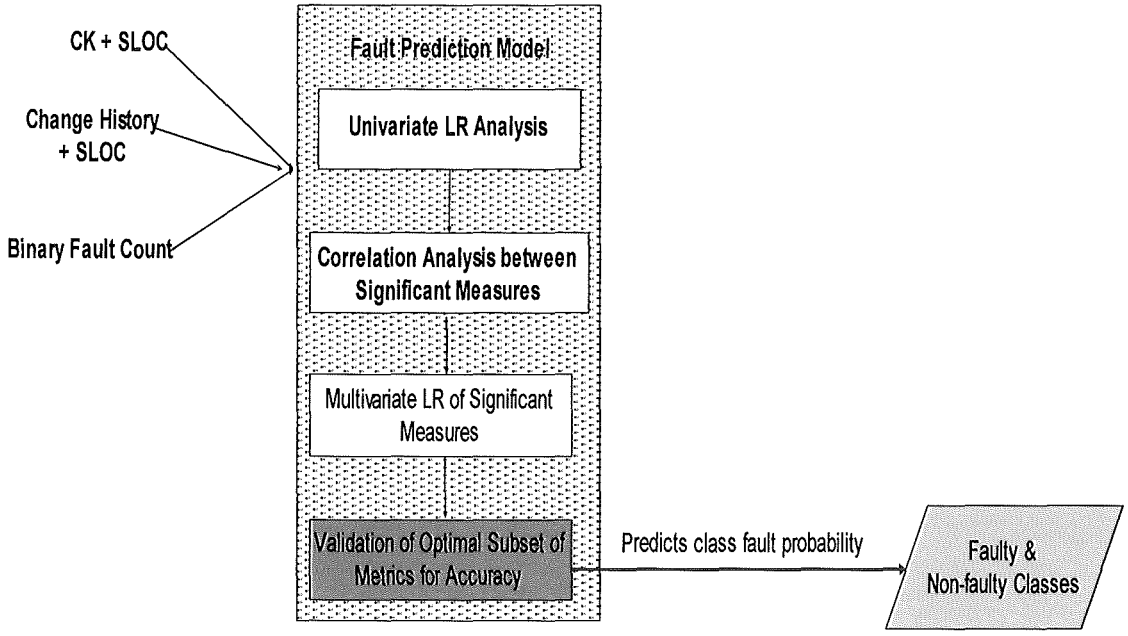


Figure 7.5: Fault Prediction Model

7.8.1 Logistic Regression Analysis

In this research, we will construct the LR model using the experience we got from both SLR performed in [18] and the assistance from an expert in the domain of statistics. LR model is different from other regression models like linear regression due to special situations that are involved in modeling. In the context of this research, the model will serve two main purposes:

1) Firstly, it will provide information about the relationship between the fault-proneness and individual class measures while other class measures are held constant. In other words, it provides us with the unique contribution of individual measures by estimating their regression coefficients.
2) It predicts which class will be risky (more fault-prone) when changes are made, allowing verification and validation activities to be focused more effectively.

With these goals, the central focal point is on the latter goal. This is to predict which classes affected by changes will result in failure after change implementation. We might not be interested in knowing the number of faults a class will have but their probabilities of being fault-prone is very important.

7.8.2 Binary Logistic Regression Model

Binary LR is one of the most widely used statistical techniques to model the relationship between certain internal products attributes and their external quality attributes such as fault-proneness. LR is the best statistical technique for building a prediction model [18]. (See Figure 7.3a) It predicts

the likelihood for an event to occur such as fault detection and is used to build predictive models when the dependent variable, Y is binary. That is, it can take on only one of two different values (0 and 1). (See Fig. 7.6) Y = 1 means that the resultant class has at least one fault and Y = 0 indicates the class is not faulty. In this model, Y is the measure of fault-proneness of a class.

Suppose that X_1, X_2, X_3,..., X_n are the independent variables and ***Prob***(Y = 1|x_1, x_2, x_3. . ., x_n) represents the probability that Y = 1 when $X_1 = x_1$, $X_2 = x_2$, $X_3 = x_3$. . ., and $X_n = x_n$. Then, LR model assumes that ***Prob***(Y = 1|x_1, x_2, x_3. . ., x_n) is connected to x_1, x_2, x_3. . ., x_n as shown in equation (7.1). ***Prob*** is the conditional probability which indicates the probability that a fault is found in a class, as a function of the class's change history and structural properties.

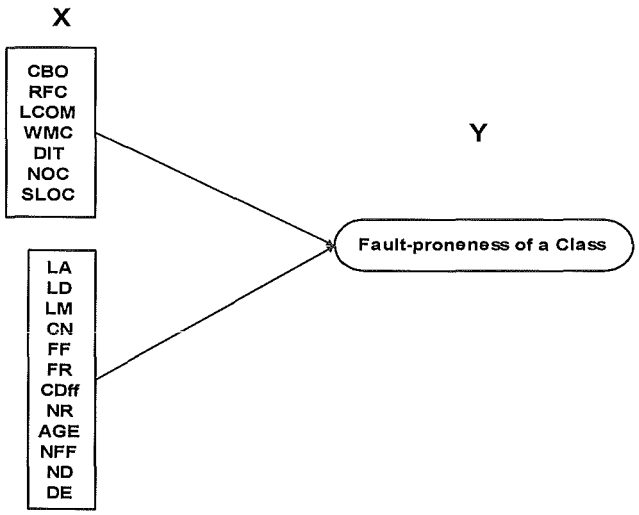


Figure 7.6: LR Model for Fault-proneness Prediction

Thus, the general format of BLR model is given by:

$$\boldsymbol{Prob}(Y = 1|x_1, x_2, x_3 \ldots, x_n) = \frac{e^{\omega + \lambda_1 x_1 + \lambda_2 x_2 + \cdots + \lambda_n x_n}}{1 + e^{e^{\omega + \lambda_1 x_1 + \lambda_2 x_2 + \cdots + \lambda_n x_n}}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots 7.1$$

In this thesis, multivariate LR and univariate LR are employed. Multivariate LR (MBLR) is used to build the prediction model for classes' fault-proneness. In this case, several metrics considered to be related to fault-proneness based on univariate and correlation analysis are used in combination to predict fault-proneness of classes. See equation (7.1) where x_1, x_2, x_3. . ., x_n are the predictors - software metrics. On the other hand, the univariate LR (UBLR) model, is a special form of MBLR involving only a single independent variable, X. In this case, UBLR is used to model the relationship that exists between the dependent variable and the individual independent variable, designed to test the stated hypothesis in order to confirm their significance with respect to fault-proneness.

The format for UBLR is given by:

$$\boldsymbol{Prob}(Y = 1|x) = \frac{e^{\omega+\lambda x}}{1+e^{\omega+\lambda x}} \quad \text{.....7.2}$$

In this research, SPSS software tool is employed to compute the model parameters. With this this software tool, the statistics which are of the essence are discussed in subsequent subsections.

7.8.3 Reported Statistics

The various statistics which are essential for the evaluation of results of the model are discussed in this section.

7.8.3.1 Estimated Regression Coefficients

In equation 7.1 and 7.2, ω and λ_is parameters (where i=1, 2,...n) are the estimated regression coefficients which are obtained through the maximum log-likelihood [22]. ω parameter is the **Y** intercept and λ_i parameter is used to show the extend of impact or contribution of each individual variable (predictor) on the model. The degree of impacts will depend on the value of λs. *The larger λ value is, the greater or stronger the impact of the individual predictor of the probability of a class's fault being detected.* This is largely determined by the sign associated with the coefficient (i.e. + positive, and - negative).

7.8.3.2 Statistical Significance

Statistical significance, **ρ(sig)** is used to show the significant level of the coefficient, λ which gives an understanding into λ's accuracy. In the context of this research, though, there exists is no standard on the performance of prediction that specifies software quality as good enough, we believed the choice of a good threshold should be dependent on the software product type and the resources available. Thus, we chose a significance threshold value of $\rho = 0.05$ or 5% probability. The validity of this choice stems from the fact that 5% probability has always been used in previous studies to determine a significant predictor. *In this case, the larger the **ρ(sig)**, the lesser the contributive impact of the independent variable.*

7.8.3.3 R-square Statistics

The R-square coefficient otherwise called $\mathbf{R^2}$ in the SPSS is a descriptive measure of the goodness-of-fit. It is defined by Cox & Snell [127]. Though $\mathbf{R^2}$ exists also in ordinary least square regression, $\mathbf{R^2}$ in LR has a different formula. $\mathbf{R^2}$ is considered a supplementary measure to other more useful evaluative measures in LR and is the proportion of the variance in the dependent variable that is explained by the variance of the independent variables. *In this case, the higher the R^2 value, the higher the model's independent variables effect would be and the more accurate the model.* However, a high value is always rare in LR [127]. In this research, Cox and Snell's $\mathbf{R^2}$ is used.

$\mathbf{R^2}$ statistic is defined formally by:

$$R^2 = 1 - \frac{LL(\omega,\lambda)}{LL(\omega)} \quad \text{.....7.3}$$

Where LL (ω,λ) is the maximum log-likelihood estimation with significant independent variables included in the model. LL (ω) is the log-likelihood estimation with only the constant (intercept) used.

7.8.3.4 Odds Ratio

Odds ratio constituteS a frequently used measure to compute the degree of association between the *dependent* and *independent* variables in the LR model. It is obtained by dividing the probability of the event by the probability of the non-event. In the context of this research, an event is the probability of detecting a faulty class and non-event is the probability of detecting non-faulty class.

Odds ratio is defined by:

$$\theta = \frac{Prob(x)}{1-Prob(x)} \quad \text{.....7.4}$$

Where $\boldsymbol{\theta}$ is the odds ratio and ***Prob (x)*** is the probability. In Eqn. (7.4), the odds that Y = 1 of detecting a fault in a class signifies the ratio of the probability of not detecting a fault in a class, given by 1 – probability of detecting a fault. In this case, it gives a measure of how Y=1 for X is affected by a unit change in the independent variable, X. For instance, if the odds ratio has 2 as a value, it indicates that the dependent variable, Y is multiplied by 2 when there is one unit increase of the independent variable, X. However, in the literature, [11] advised that it should not be used to compare the relative extent of the relationships between different metrics and fault-proneness since different metrics have different units associated with them. In the SPSS, odds ratio is computed from the coefficients, $\boldsymbol{\lambda}$ and $\boldsymbol{\lambda}$ as $\mathbf{e^{\lambda i}}$ or $\mathbf{Exp(\lambda)}$.

7.8.3.5 Maximum Likelihood Estimation

Maximum likelihood estimation is a statistical method that is used to quantify the coefficients ($\boldsymbol{\omega}$ and $\boldsymbol{\lambda}$) of a model. If the likelihood function is given by **L**, the maximum likelihood estimation is used to compute the coefficients that make the log of the likelihood function given by Log (L) as large as possible. In this case, the larger the coefficient value the greater the impact of the corresponding predictors (independent variables) on the fault detection probability. In the SPSS statistical tool, the maximum likelihood estimation is computed with *-2 Log Likelihood. The smaller the statistic the better the model will be.*

7.9 Fitting the Model

This section describes how the model will be fitted. This is aimed at quantifying the relationship between the probability of a "faulty class" outcome and the independent variables X_1, X_2, ..., X_k. Based on the Eqn (7.1), the LR model in this thesis will be formulated based on likelihood measured by probability and odds. In this case, the Odds ratio shown in Eqn (7.4) is used. This is achieved via the transformation of ***Prob(x)*** instead of fitting the model for ***Prob(x).*** The log transformation of ***Prob(x)*** or *Logit* ***Prob(x)*** is used by letting log ***Prob(x)*** be a linear function of x. This transformation is to ensure that changing predictors multiplies the probability by a fixed amount, a situation that is not obtainable in linear regression. If the natural logarithm of θ is taken on Eqn (7.4), the result will be:

$$Logit(Y) = In\left(\frac{Prob(x)}{1-Prob(x)}\right) = \omega + \lambda_1 x_1 + \lambda_2 x_2 + \cdots + \lambda_k x_k \quad \text{.....7.5}$$

Therefore, the predicted class fault-proneness will be:

$$\textbf{Logit (Y)} = \omega + \lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_k x_k \quad \text{.....7.6}$$

$$\rightarrow \quad \textbf{\textit{Logit}}(Y) = \omega + \sum \lambda_i x_i$$

This produces a linear model on the logit scale, where, Logit(Y) is the natural logarithm of the measure of fault proneness or ***Prob***(x), ω is the Y intercept or constant and λ_i (i=1,2...,k) the regression coefficient of the individual measures considered as good predictors used in the model while X_i (i=1,2,...,k) are the software measures (OO metrics and change history) used in the model.

Thus, a transformation of equation (7.6) will result back to Eqn.(7.1 or 7.2) and can be written in a slightly different way as:

$$\textbf{\textit{Prob}}(Y = 1|x) = \frac{1}{1+e^{-(\omega+\sum\lambda x)}} \quad \text{.....7.7}$$

As the goal is to classify a class as fault-prone (1) or not fault-prone (0), in order to curtail the misclassification rate, a class is predicted as faulty, Y = 1 when $p \geq 0.5$ and not faulty = 0 when $p < 0.5$.

As shown in Eqn (7.6), the binary value, Y = 1 whenever $\omega + \lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_k x_k$ is non-negative, and 0 otherwise. Thus, LR gives a linear classifier. However:

- LR only classifies a class as faulty or not faulty and does not tell the number of faults per class,

- Also, it does not specify if it is high or low, but with the cutoff point, we will be able to know which class has the highest probability of fault-proneness. In this case, the higher the failure-proneness of a class, the higher is the probability of it failing in the field.

7.9.1 A Typical Example

As an illustration, if a model is given by:

Predicted Logit of Fault-proneness = 0.5430 + (0.0027)* CBO +(0.0036)*SLOC + (0.0045)*WMC, that predicts the fault-proneness of a particular class, say **A**. Where, 0.0027, 0.0036 and 0.0045 are λ for CBO, SLOC and WMC respectively, where CBO=10, SLOC= 23, and WMC = 14 and ω =0.5430.

Therefore, the logit (Y) will be computed as:

Logit(Y) = 0.543 + (0.0027*10) + (0.0036*23) + (0.0045*14)

= 0.027+0.0828 + 0.063 = 0.1728

Logit (Y) = 0.1728

Inserting this into Eqn (7.6)

$$\textit{Prob}(Y = 1|x) = \frac{1}{1+e^{-(0.1728)}}$$

$$= \mathbf{0.5431\ (54\%)}$$

This indicates that, the model predicts the class **A** will have 54% chance of being fault-prone or failing in the field if a proposed change is made on it. The section that follows explains the various approaches involved in the selection of optimal subset of measures (OO metrics, size and change) which are significant and are considered good predictors of fault-proneness prediction.

7.10 Metrics Selection Approaches

The various steps involved in selecting optimal sets of metrics or measures considered as good predictors of fault-proneness to be used in constructing the prediction model is discussed in this section. The different data analysis methods used as captured in Figure 7.5 are as follows:

i. Do UBLR analysis in order to evaluate or assess each individual metric's relationship with fault proneness.

ii. Do correlation analysis between all identified significant metrics in (i) to identify the metrics that are highly correlated with each other.

iii. Do MBLR on the identified metrics in (ii) to determine a subset of metrics for an improved performance of fault prediction models.

iv. Evaluate the results of the subset metrics to estimate the overall model accuracy using confusion matrix criteria.

7.10.1 UBLR Analysis

In the thesis, the first step towards the metric selection is to perform a UBLR analysis on all the metrics involved. This is achieved by using the CK metrics, SLOC and the change data each as the independent variable in the model in turn. The basis is to evaluate each individual metric for fault-proneness. In this case, the dependent variable is the measure of faults in the system. In order to determine the level of significance of each individual metric, the following LR model parameters are used:

i. λ & ω - Regression coefficient which indicates the extent of correlation of each metric with fault proneness.
ii. Significance level (p- value) at p = 0.01 as a cutoff indicating the significance of each correlation,
iii. The odds ratio given by Exp(λ), and
iv. $\mathbf{R^2}$ – R-square statistics. The bigger the value of $\mathbf{R^2}$ the better the independent variable explained the dependent variable in the model.

The output of this step will result in a subset of the metrics that are significantly related to fault-proneness for fault prediction. A metric is significantly related to class fault-proneness if its ρ-value is less than or equal to 0.05, otherwise it is not. This ρ-value for each metric tells us if the metric is a significant predictor of the dependent variable.

7.10.2 Correlation Analysis

After checking each metric for association with fault-proneness, the next step is to determine the correlation between the metrics that were found significant. To achieve this, a Spearman rank correlation between all the significant metrics was conducted to establish the metrics that are positively or negatively correlated with each other. This is achieved by taking one measure at a time as dependent variable while the others are used as independent variables. This analysis is because most metrics are not completely independent. In this case, the existence of strong correlations among some metrics will exclude some metrics as candidates for the MBLR, a situation called multi-collinearity [87][93]. Multi-collinearity is used to describe the degree to which one variable effect can be predicted by the other variables [87], and this makes the interpretation of the model very difficult.

However, this can be solved by checking the performance of each metric individually as well as in combination with other metrics for fault prediction and a metric or metrics set with good performance is selected. The process is repeated until all highly correlated independent metric are dealt with. This is done at the 95% confidence level (p-value $\leq$ 0.05). For instance, if CBO-SLOC and WMC-SLOC are found to be highly correlated, one conclusion could be that CBO and WMC

are good indicators of class complexity and SLOC is not needed for measuring class size separately in the model. The magnitude and signs associated with the correlation results are therefore used to identify the strength and characteristics of the relationship between the software measures and failure-proneness. Accordingly, positive correlation indicates significance and negative correlation indicate insignificance. At the end of this analysis, the outcome is a metric subset that is non-redundant and significantly good for fault prediction.

7.10.3 MBLR Analysis

The elimination of the multi-collinearity among the metrics in the model does not guarantee the best set of independent variables in the MBLR models. The reason is that it is possible that there still exist some metrics in the subset that depend strongly on others. In this case, they have to be reduced via a MBLR model. MBLR analysis involves determining a best possible subset of the metrics that can predict fault-proneness. This analysis is performed using metrics in combination and constructing several models. In this work, for each model, significantly good subsets of metrics are selected using the stepwise LR procedure to build the prediction models. Each of the steps variables to enters and leaves model. The resultant analysis is an optimal subset of the metrics that is significant to predict fault using Eqn (7.6).

7.10.4 Model Validation

After the selection of the optimal subset of the metrics that are significant to predict faulty classes, the final step is to construct prediction models with those metrics in order to examine their effectiveness in fault-proneness prediction. The fundamental goal of this step is to help us estimate the overall predictive accuracy of the metrics and not the identification of the best fault prediction method.

7.11 Model Evaluation Criteria

Logistic regression proffers a classification model that tries to predict whether a class will have at least one fault or not. However, for LR model to be able to predict correctly and accurately which classes truly have faults or not depends on the choice of optimal subsets of metrics used or selected through both UBLR and MBLR. As the task of the MBLR is to choose subsets of predictors that yield the best classification which will be used to construct the prediction model, in order to ascertain the predictive power of these metrics, the model needs to be evaluated for goodness-of-fit of the risk that a particular change will pose. In this case, the predicted fault-proneness of classes to their actual or true fault-proneness can be compared.

To assess the model, the confusion matrix criteria is used [87][107] which is composed of several measures. The choice of confusion matrix stemmed from the fact that it is popular and previous researches have used it to evaluate their models' predictive effectiveness with respect to LR [107].

Table 7.2: Confusion Matrix

		Actual Faults Observed	
		True	**False**
Model Predicted Class Faults	**Positive**	True Positive (TP)	False Positive (FP) (Type I error)
	Negative	False Negative (FN) (Type II error)	True Negative (TN)

The confusion matrix is captured in Table 7.2 and defines the statistical measures that follow. The outcomes of the classification are defined with respect to positive (P) or identified and negative (N) or rejected criteria as follows:

1) True positive (T_P): Faulty classes correctly identified as faulty
2) False positive (F_P): Non-faulty classes incorrectly identified as faulty
3) True negative(T_N): Non-faulty classes correctly identified as non-faulty
4) False negative (F_N): Faulty classes incorrectly identified as non-faulty

Based on these notations, it is possible to straightforwardly identify classes which were misclassified leading to Type I and II errors. F_P classification will lead to Type I error while F_N will give rise to Type II error. However, in this research, the aim is to ensure that both Type I and II errors are reduced to the barest minimum or completely eliminated in the model. The statistical measures used are among the most frequently used measures for quantifying the predictive effectiveness of classification models like LR. The measures are defined as follows:

7.11.1 Sensitivity

Sensitivity is a statistical measure that is used to evaluate the performance of the prediction model. It computes the proportion of faulty classes that have been correctly identified or classified as faulty classes. In other words, it is the ratio of the number of classes which are correctly classified as fault-prone to the total number of classes that have faults in them. Sensitivity is sometimes referred to as T_P rate or *recall* rate.

Formally, sensitivity is defined as:

$$\textbf{\textit{Sensitivity}} = \frac{T_P}{T_P + F_N} \quad \text{.....7.8}$$

Having sensitivity or a recall value of 100% means that every faulty classes that would have led to field failure was classified as being faulty in order to focus effective testing activities on them before or after a change is implemented.

7.11.2 Specificity

Specificity or T_N is another measure used to evaluate the prediction model's goodness-of-fit. In this case, specificity computes the proportion of classes identified as not faulty that have been correctly classified as non-faulty classes in the system. In other words, it is the ratio of the number of classes correctly identified or classified as non-faulty to the total number of classes that are non-faulty.

Formally, Specificity is defined as:

$$\boldsymbol{Specificity} = \frac{T_N}{F_P + T_N} \quad \text{.....7.9}$$

Given the measures of *sensitivity* and *specificity*, it is important that they have values that are as high as possible. This is because, for instance, a low *specificity* value would indicate that there are several non-faulty classes that were classified as faulty. In this case, resources will be wasted in channeling validation and verification activities on such class as before or after changes are made. Having a low *sensitivity* would indicate that several faulty classes were being classified as non-faulty. Consequently, several fault-prone classes would be passed on to the subsequent releases to the customer which in turn would increase the risks of field failures. However, regardless of the measure, the fact remains that penalties that originate from them may be costly field failures or costly fault correction in later phases.

7.11.3 Accuracy

Accuracy is the statistical measure defined as the ratio of number of classes that are correctly classified as faulty or non-faulty to the total number of classes that were observed or identified to be faulty or non-faulty in the OO software system. In other words, it is the percentage of classes correctly classified as either faulty or non-faulty in terms of T_P and T_N.

We formally defined accuracy as:

$$\boldsymbol{Accuracy} = \frac{T_P + T_N}{T_P + F_P + T_N + F_N} \quad \text{.....7.11}$$

If a computed value for accuracy is 1, it is considered to be the best classification which shows that the classification by the model was perfect with no single error.

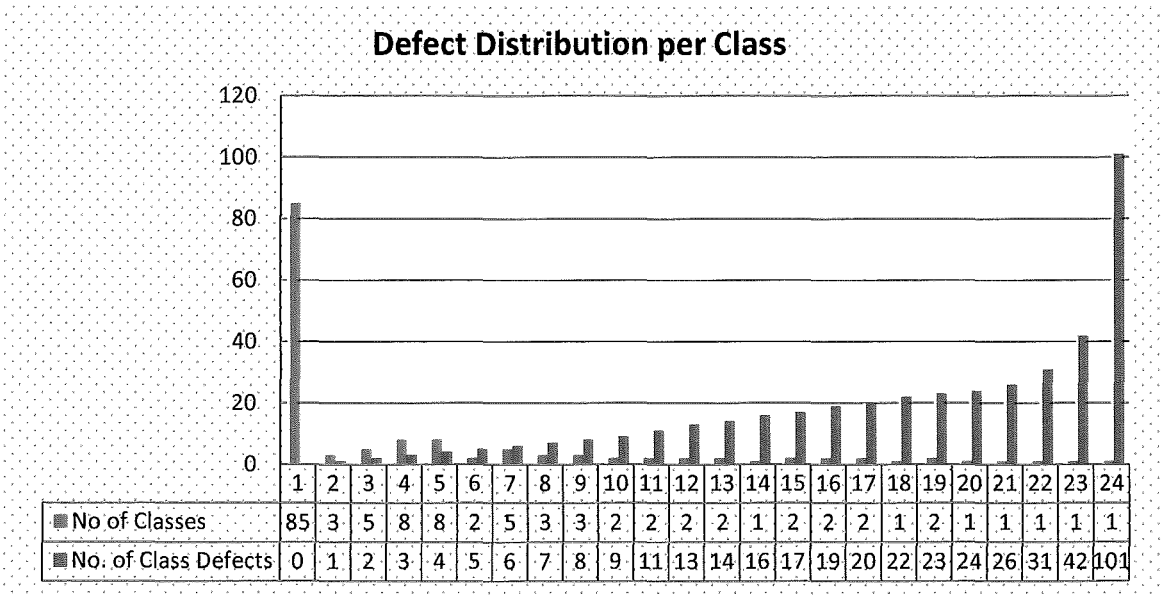
All the measures are evaluation criteria for the prediction model. For these measures to be correctly interpreted, it is important that they should be presented as percentages. They are all reported in the SPSS statistical tool. In addition, the cutoff, $P_c = 0.5$ is used.

7.12 Fault Prediction Model Evaluation

This section presents the evaluation of the fault prediction model. This study is a semi-replicated study and the intention is not to directly validate the influence of product metrics on the final quality of software. In this case, no hypothesis will be tested. In addition, since no repository has been maintained for process metrics, at this point only product metrics, CK + SLOC metric will be used. The study goal is only to evaluate the effectiveness of the prediction model using the dataset discussed below.

7.12.1 Empirical Data Description

This study used the public domain data set KC1 collected from the NASA Metrics Data Program (NASA, 2004) [132]. This data was collected and validated by the Metrics Data Program (MDP, 2006) and stored in the NASA data repository. Accordingly, the KC1 data was gathered from a storage management system developed in C++ programming language for receiving/processing ground data. During the course of the projects, fault data for KC1 was maintained and collected starting from the beginning of the project for 5 years. More details can be found in [132]. The storage management system consists of 145 classes having 2,107 methods with 40 KSLOC. KC1 offers static software metrics at both class-level and method-level. However, only static metrics at the class-level are of interest in this research. The values of 7 metrics were computed including CK metric suit and SLOC at the class-level for our analysis. Fault data were collected by associating methods to classes as well as their reported defects. Consequently, out of the 145, 60 classes were found to be faulty while 85 are not faulty. We captured the distribution of the defects in each class in Figure 7.7.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
No of Classes	85	3	5	8	8	2	5	3	3	2	2	2	2	1	2	2	2	1	2	1	1	1	1	1
No. of Class Defects	0	1	2	3	4	5	6	7	8	9	11	13	14	16	17	19	20	22	23	24	26	31	42	101

Figure 7.7: Number of Defects per Class

The rationale for using NASA KC1 dataset is as follows:

1. The size of the students' projects we used in previous experiments was too small to collect the fault and product metrics from them to be used in the model. This is because the objective is to support CIA in large-scale software.
2. NASA KC1 dataset was collected from a real-world system stored in a public domain that is recognized worldwide and has been used for empirical analysis by renowned researchers such as [81][86][94][100].

7.12.2 Methodology and Analysis Results

This section presents the analysis of the empirical validation carried out to predict class fault proneness and failure. We employed the model discussed in Section 7.8; logistic regression, LR in this analysis. It followed strictly the steps discussed in Section 7.10, UBLR ans MBLR models to discover the relationships between the different OO + SLOC metrics and the fault proneness of the classes. The prediction models were applied to all the 145 classes.

7.12.2.1 Descriptive Statistics of Metrics

Descriptive statistics play an important part in analyzing and quantitatively describing the main features of a collection of research data. They help in reducing research data to a form that can be read easily, from which conclusions can be drawn and which can be used for further analysis. In this thesis, the essential statistics measures used for comparing the different metrics of interest are the minimum, maximum, mean, median, and standard deviation as captured in Table 7.3

Table 7.3: Metrics Descriptive Statistics

Metrics	N	Min	Max	Mean	Std. Dev.
CBO	145	0	24	8.32	4.270
LCOM	145	0	100	68.72	18.362
NOC	145	0	5	.21	.747
RFC	145	0	222	34.38	26.493
WMC	145	0	100	17.42	14.597
DIT	145	0	6	1.00	1.149
SLOC	145	0	2313	211.25	190.033
Fault Content	145	0	101	4.61	10.841

As we can see from Table 7.3, class size is measured with respect to lines of code that vary between 0 and 2313. SLOC has the highest mean indicating the biggest measure per class collected in the KC1 dataset. In the same vein, LCOM measure has higher values (100) in KC1 data set compared to CBO (24). This shows the quality of the design with high cohesion low

coupling. Finally, DIT and NOC values are very low, indicating inheritance is not much used in all the systems.

7.12.2.2 Analysis Results

This section presents the result of the logistic regression analysis. Faults data were used as the dependent variable while the product metrics are used as the independent variables. Based on the steps highlighted in Section 7.9, we performed UBLR first, followed by correlation analysis between the metrics and lastly, the MBLR. The rationale for these statistical analysis steps is to select an optimum subset of metrics that will be used to construct our prediction model.

A. UBLR Analysis

The results of the UBLR for predicting class fault proneness is captured in Table 7.4. The table presents the regression coefficient (λ), the statistical significance ρ- value, odds ratio (Exp(λ)), and R^2 statistic for each measure. Classes were treated on the basis that a class is faulty if it had at least one fault, otherwise it was not faulty. Metrics were selected based on positive λ, ρ-value ≤ 0.05 and Exp(λ) > 1. The cutoff value of 0.5 was used for the classification.

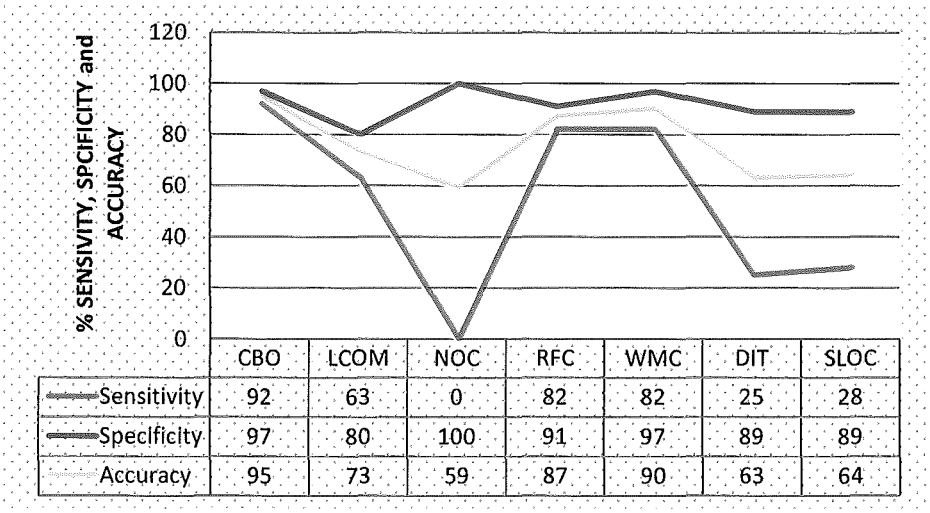
In Table 7.4 and based on the selection criteria, NOC metric was not found to be significant while LCOM metric has a negative coefficient and the odds ratio is less than 1. This shows that the LCOM metric is related to fault proneness but in an inverse manner. CBO Metric has the highest R^2 indicating it is the best predictor followed by WMC metric. Based on these results, R^2 values are considered more important than the ρ-values as they indicate the correlation strength.

Table 7.4: UBLR Results

Metrics	λ	ρ-value	Exp (λ)	R^2
CBO	2.040	0.000	7.688	0.649
LCOM	-0.091	0.000	0.913	0.290
NOC	0.009	0.969	1.009	0.000
RFC	0.127	0.000	1.135	0.506
WMC	0.171	0.000	1.187	0.443
DIT	0.585	0.002	1.794	0.081
SLOC	0.006	0.010	1.007	0.060

Thus, CBO, RFC, and WMC are found to be more significant with respect to their R^2 value. This is followed by SLOC and DIT. In these results, CBO is considered to be the most effective at that level of significance. The result of NOC insignificance is in line with [86][94][100]. We therefore dropped LCOM and NOC for further analysis. Figure 7.8 presents the sensitivity, specificity and

the accuracy of each metric. Table 7.4 shows that CBO, RFC and WMC have the highest sensitivity and specificity as well as accuracy. This is followed by LCOM, DIT and SLOC.



	CBO	LCOM	NOC	RFC	WMC	DIT	SLOC
Sensitivity	92	63	0	82	82	25	28
Specificity	97	80	100	91	97	89	89
Accuracy	95	73	59	87	90	63	64

Figure 7.8: Sensitivity, Specificity and Accuracy of Individual Metric

B. Correlation Analysis

Table 7.5 shows the result of the Spearman's rank correlation among metrics. The correlation coefficient value was based on Hopkins [133] classification where values between 0.5 and 0.7 are large, 0.7 to 0.9 are very large while 0.9 to 1.0 are almost perfect. In addition, this classification have been used by [93][133] in their studies. The threshold value is 0.5 and the significance level is $\rho \leq 0.01$ level. The results in Table 7.5 show that WMC, DIT, SLOC metrics are correlated with RFC metric, and RFC, WMC and CBO metrics are correlated with SLOC metric.

Table 7.5: Metric Correlations

Metrics	CBO	RFC	WMC	DIT	SLOC
CBO	1				
RFC	.691**	1			
WMC	.619**	.529**	1		
DIT	.255**	.152	.367**	1	
SLOC	.171*	.312**	.110	.064	1

**. Correlation is significant at the 0.01 level (2-tailed).
*. Correlation is significant at the 0.05 level (2-tailed).

In general, the results indicate that the metrics are not completely independent or redundant with each other and are significantly good for fault prediction. However, the correlations among CBO, RFC, and WMC and between RFC and WMC are very strong. Therefore, we selected CBO, RFC, WMC, DIT and SLOC to be included in the MBLR model.

C. MBLR Analysis

In this section, we present the results of the MBLR used to find the combined effect of the CK + SLOC metrics on fault proneness. The analysis was aimed at selecting metric subsets that yield the optimum classification in the model. In this case, we used the forward stepwise procedure and the results obtained are good in terms of R^2 and log likelihood statistic. We used a classifier at threshold = 0.5 as cutoff value, and all the 145 classes were used as training data. Due to the multi-collinearity effect, we have three sets of predictors as potential metrics for the construction of the fault prediction model. Table 7.6a provides the regression coefficient (λ), statistical significance (ρ-value), odds ratio (Exp(λ)), R^2, the -2 Log likelihood and the constant (ω) for metrics included in the model while Table 7.6b presents the result of the classification.

Table 7.6a. MBLR Results for Metric Set I

Metrics set I	λ	ρ-value	Exp (λ)
CBO	2.867	0.005	17.586
RFC	0.195	0.024	1.215
WMC	0.088	0.018	1.092
R^2	0.711		
- 2 Log likelihood	16.681		
Constant (ω)	-36.938		

Table 7.6b: Classification Results for Metric Set I

LR Model		Predicted Fault		Sensitivity (%)	Specificity (%)	Accuracy (%)
		Non-Faulty	Faulty			
	Non-Faulty	85	2	97	98	97
Fault	Faulty	2	58			

Table 7.7a provides the regression coefficient (λ), statistical significance (ρ-value), odds ratio (Exp(λ)), R^2, the -2 Log likelihood and the constant (ω) for metrics included in the model while Table 7.7b presents the result of the classification.

Table 7.7a. MBLR Results for Metric Set II

Metrics set II	λ	ρ-value	Exp (λ)
SLOC	0.008	0.023	1.008
WMC	0.173	0.000	1.189
R^2	0.464		
- 2 Log likelihood	106.227		
Constant (ω)	- 4.857		

Table 7.7b: Classification Results for Metric Set II

LR Model		Predicted Fault		Sensitivity (%)	Specificity (%)	Accuracy (%)
		Non-Faulty	Faulty			
Fault	Non-Faulty	82	3	82	97	90
	Faulty	11	49			

Table 7.8a provides the regression coefficient (λ), statistical significance (ρ-value), odds ratio (Exp(λ)), R^2, the -2 Log likelihood and the constant (ω) for metrics included in the model while Table 7.8b also presents the result of the classification.

Table 7.8a. MBLR Results for Metric Set III

Metrics set III	λ	ρ-value	Exp (λ)
CBO	2.938	0.005	18.878
RFC	0.200	0.027	1.221
DIT	2.214	0.021	9.152
R^2	0.709		
- 2 Log likelihood	17.605		
Constant (ω)	- 37.124		

Table 7.8b: Classification Results for Metric Set III

LR Model		Predicted Fault		Sensitivity (%)	Specificity (%)	Accuracy (%)
		Non-Faulty	Faulty			
Fault	Non-Faulty	84	1	98	99	99
	Faulty	1	59			

7.12.3 Model Evaluations

In this section, we evaluate the predicted model based on sensitivity, specificity and accuracy. Table 7.9 presents the summary of the predicted model. A dash on the table shows that the metric was not selected for that model.

Table 7.9: Model Predictors

	Predictors (λ)					Constant (ω)
Predicted Model	**CBO**	**RFC**	**WMC**	**DIT**	**SLOC**	
M_1	2.867	0.195	0.711	-	-	-36.938
M_2	-	-	0.173	-	0.008	-4.857
M_3	2.938	0.200	-	2.214	-	-37.124

As shown on Table 7.6a, Table 7.7a and Table 7.8a, the maximum likelihood estimate of the models, M_1 – M_3 which is a measure of how poorly the model predicts the fault proneness of classes are very encouraging. This shows that the models are good at prediction since the smaller the statistic, the better the model. Accordingly, M_1 has a maximum likelihood value of 16.681, M_2 106.227 and M_3 17.605. Also on the predicted model are the Cox & Snell R^2 values. The values were very high though it is difficult to achieve a value of 1. This shows that the models are accurate in the prediction, since the higher the R^2 values, the higher the effect of the models. Accordingly, the R^2 value are all positive where M_1 is 0.711, M_2 0.464 and M_3 0.709. In this case, M_1 and M_3 have the highest R^2 value which indicates the strength of the explanatory variable in detecting the probability of a fault in a class. In general, all metrics are significant in their respective model.

In terms of sensitivity, specificity and accuracy, the values are very high indicating that in the model, several non-faulty and faulty classes were correctly classified as non-faulty and faulty classes respectively. In this case, no resources will be wasted in channeling validation and verification activities on such classes when changes will be implemented on them. In addition, no or only few fault-prone classes would be passed on to the subsequent releases to the customer which in turn would reduce the risks of field failures. In the predicted models, M_1 – M_3, we had 97% sensitivity, 98% for specificity and 97% accuracy for M_1. For M_2 we had 82% sensitivity, 97% specificity and 90% accuracy while M_3 had 98% sensitivity, 99% specificity and 99% accuracy. In general, the high accuracy of the model indicates that lesser the effort will be spent on detecting fault-prone or fault-free classes with improvement in the efficiency.

7.13 Model CIA Application

In this section, we present the application of the prediction model during the course of software change impact analysis. As a core goal of this thesis, we aim to support CIA activities in large-scale software applications in order to identify which set of classes are more likely to have faults or fail in the field if they are changed. By so doing, validation and verification efforts will be channel to those classes which are likely to have faults or fail in order to reduce impending risks.

Based on the KC1 dataset we used in the evaluation of this model, we assumed that major maintenance is going to be carried out on the current version, which would result in the next new version. What needs to be done is to follow the CIA framework shown in Figure 3.4 of chapter 3. The maintainer has to analyze the change request and then identify the subset of classes that will be affected by the change request by using either the CIA technique discussed in this thesis or any other CIA technique that is known to the maintainer. This will form the total impact set (TIS) for the change. Based on the metrics and the fault data collected, the maintainer can then compute the fault-proneness of each class identified to be affected by the change using any of the prediction models $M_1 - M_3$. For this illustration, we chose M_3 since it yielded the highest accuracy rate of 99%. The model is therefore given by:

$$(1/\text{Class Fault Proneness}) = (-37.124 + 2.938(\text{CBO}) + 0.2(\text{RFC}) + 2.214(\text{DIT}))$$

A class is considered fault prone if the risk is more than 50%. That is, the risk rate $\geq$ cutoff value (0.5). However, in order to compute the risk associated with each class, we developed a novel tool called Class Change Recommender or simply CCRecommender.

7.13.1 Class Change Recommender Tool

CCRecommender is a novel and stand-alone tool developed to help software maintainers to compute the risk or fault proneness of each class affected by a change request before the change is implemented and also give advice on whether to proceed with the change or not. CCRecommender is a tool developed with java programming language, having interactive interfaces for ease of use. In order to use the tool, first we have to build the model using any statistical tool. In this case, the SPSS statistical software is used in order to compute values of the regression coefficients, ω, λ, and others. These values are then used alongside the metric values for each included metric for prediction.

In M_3 model chosen, three metrics are included, CBO, RFC and DIT. The home page of the tool is captured in Figure 7.9.

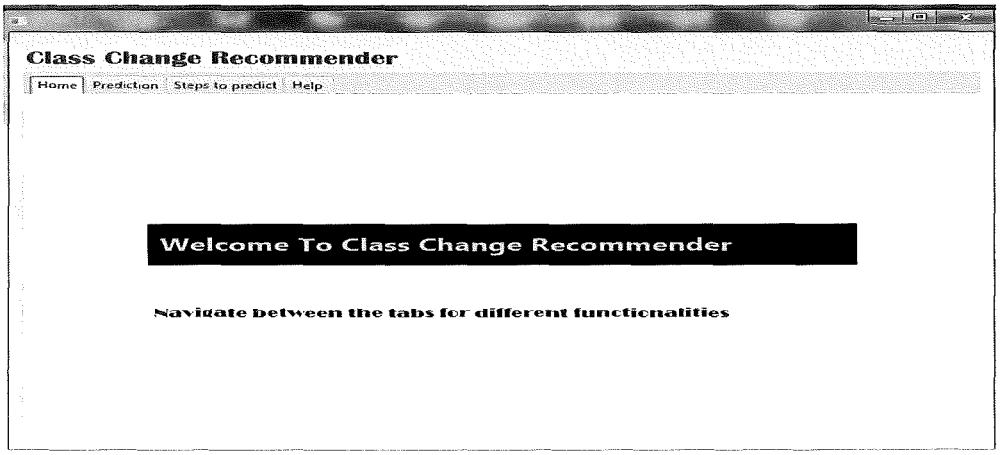


Figure 7.9: CCRecommender System

As shown in Figure 7.9, the menu bar shows the prediction menu, the steps to predict and help menus. The step to predict menu contains the different steps that will be used to predict the fault proneness of each class. Accordingly, the help menu contains the description of all the metrics and parameters that are fundamental to the model, while the prediction menu is where the actual computation takes places. (see Figure 7.10)

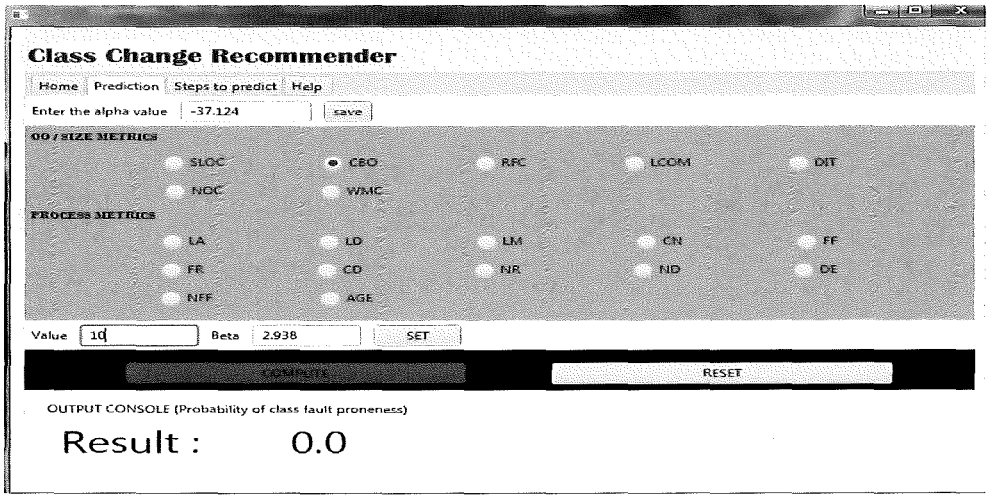


Figure 7.10: Prediction Interface of CCRecommender

Figure 7.10 presents the interface where the values of the model parameters will be entered. First is the entering of the intercept (ω) called alpha. Secondly, is the choice of the metrics included in the model and the entering of their values as well as their respective regression coefficients (λ) called beta. After entering the value and the λ value for a metric, the button "SET" is used to input the values of the next chosen metric. In this case, after inputting all the parameters values into the system, the next step is to click the button called "COMPUTE" and what appears is the risk or fault proneness value in percentage of the class.

The tool also gives modification advice based on the risk rate of the class on whether a change is recommended or not. In this case, we modeled the change recommendation based on the following cutoff probabilities:

1) If the probability is less than or equals to 20, (prob. ≤ 0.2) a GREEN circle will appear on the result, indicating the class is not fault prone and the change can be implemented. For instance, the result in Figure 7.11a, shows that the risk associated with making changes to a class that has CBO = RFC =10 and DIT = 1 is 3%.

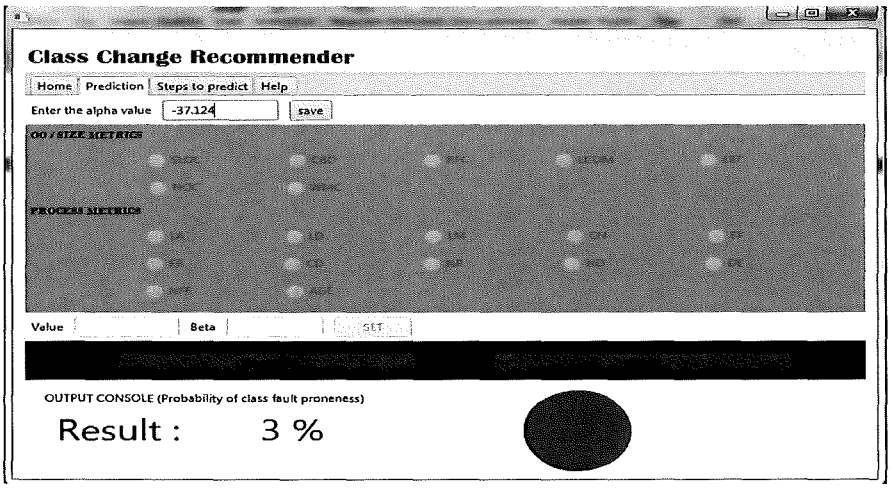


Figure 7.11a: Prediction Result for Condition 1

2) If the probability is greater than 20% but less than or equals to 50% ($0.2 \geq$ prob. ≤ 0.5), a BLUE circle will appear indicating theclass is not fault-prone but care must be taken when making changes to such a class. For example, if a particular class has CBO = 10, RFC = 10 and DIT = 1, the prediction on the system will yield the following result shown in Figure 7.11b.

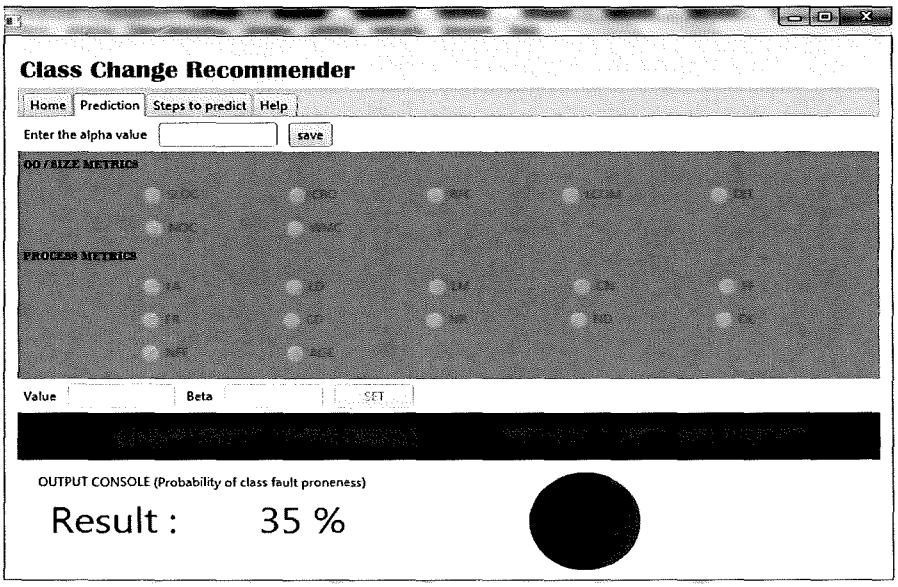


Figure 7.11b: Prediction Result for Condition 2

3) If the probability is greater than 50% but less than or equals to 70%, (0.5 ≥ prob. ≤ 0.7), a circle will appear, indicating the class is fault-prone.Before changes can be made, verification and validation activities have to be channeled to the class. Accordingly, the fault proneness probability for a class having metric values, CBO = 11, RFC = 25 and DIT = 0 will be 55% as shown in Figure 7.11c.

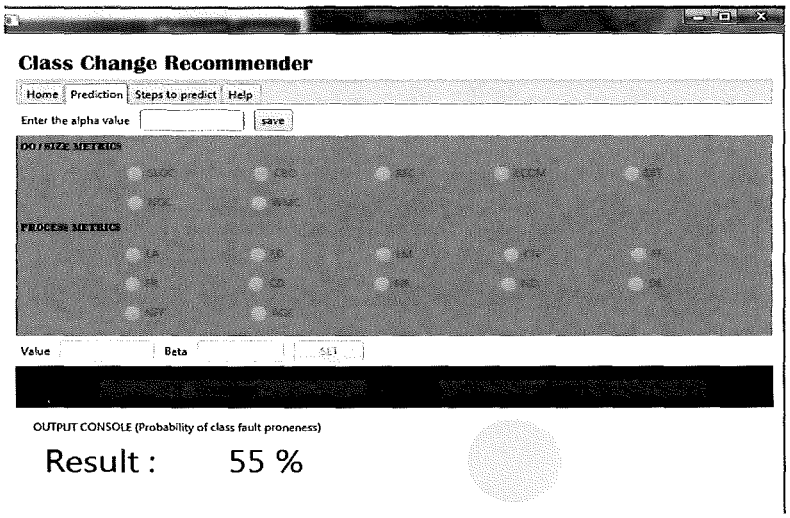


Figure 8.11c: Prediction Result for Condition 3

4) Lastly, if the probability is greater than 70%, (prob. ≥ 0.7), a RED circle will appear, indicating that the class is failure-prone. In this case, change is not recommended on such as class. This is captured in Figure 7.11d for class having CBO = 8, RFC = 36 and DIT = 5, the risk rate or fault proneness probability will be 99%.

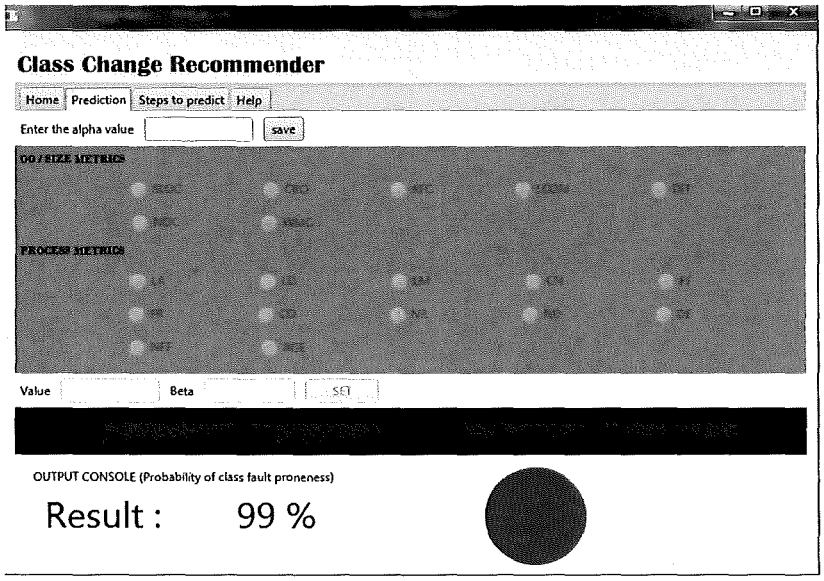


Figure 7.11d: Prediction Result for Condition 4

The above results shows the risk or fault-proneness probabilities associated with classes having metrics values of CBO, RFC and DIT. This indicates that a class with a high value of coupling, a

high number of external and internal methods and with a greater depth in inheritance tree is more likely to be fault-prone than a class with lower values of these metrics. It is important that during CIA, the maintainer needs to have this information in advance in order to know if a change is going to be risky or not, as well as to provide a mitigating plan aimed at reducing or avoiding the costly risk.

7.13.2 Threats to Validity

The study has some limitations, which we believe are not unique to our study but are also common with most empirical studies. The extent to which the results of our study can be generalized to other similar research may be questionable. This study is a kind of semi-replicated study where the KC1 data set, faults data and product metrics were not collected directly from the NASA repository, but through similar studies. In addition, other studies that have conducted similar research considered fault at different severity levels which we did not take into account. In this case, the metrics and fault data may be inaccurate. Thus, we make no claim about the accuracy of the data set, and the tools used in collecting them. However, we are confident that the fault data and the metrics values as well as their collection will be valid for more studies than ours which utilized them for the demonstration of the prediction model.

Furthermore, we do not in any way suggest the generalization of the research results to any other similar empirical studies in terms of metrics validation because our study did not test any hypothesis. The results can only be generalized if the KC1 dataset were collected directly from the NASA repository. However, the results obtained here provide guidance on how to use the model to predict fault-proneness of classes using fault data as dependent variable and OO metrics as independent variables. Also, the accuracy of the model predicted is quite high, but it is somewhat optimistic since is only used for evaluation purposes. Thus, more validations are required with different applications in order for stronger conclusions to be drawn.

7.14 Chapter Summary

Software fault prediction models provide an important opportunity to identify faulty classes earlier in the development life cycle. In the context of this thesis, it is used to identify faulty classes early during CIA. In this chapter, we have described the processes involved in extracting these important software measures, the identification of *before* and *after-release* faults, the fault prediction model technique using LR and the model evaluation criteria to assess the goodness-of-fit. In addition, the failure prediction has a high prediction accuracy, sensitivity and specificity. Therefore, we recommend them for the support of CIA during software maintenance activities of OO programs.

CHAPTER 8

Summary, Conclusions and Future Works

This chapter summarizes and concludes the thesis. The main research questions we introduced and motivated in Section 1.4 are:

> **MRQ1:** *With the complexity associated with the relationships existing in object-oriented programs, how can we perform change impact analysis that will effectively capture the relationships and reduce the impact sets for successful change implementation?*
>
> **MRQ2:** *Can we develop a change propagation framework that will predict early the failure or the risks associated with change implementation based on the predicted impact sets?*

The chapter begins with the overall summary of the thesis with respect to **MRQ1** and **MQR2**, it then progresses to the conclusions, recommendations, limitations and ends with some idea for future work.

8.1 Summary

In this thesis, the main goal rested on designing a change impact analysis framework and model for early failure prediction that will predict the impact of changes and failure in object-oriented programs. The basis was to improve the current object-oriented maintenance strategy to enhance the quality of the software while reducing the costs, efforts and the risks associated with software changes.

In order to achieve the goal of this thesis, research questions (**MRQ1** and **MRQ2**) were addressed. Firstly, we designed a framework for change propagation as an approach to assist software maintainers to perform change impact analysis in object-oriented software both in small and large systems before a change is implemented. This will assist software maintainers to determine *a priori*, which components will be truly affected by a change and which of those components will be risky to change. Identifying the potential impacts and the risk early before making a change, will go a long way to reduce the risks associated with making a costly change because the earlier the problem is identified the less the cost of solving it. The framework developed in this thesis sets the direction to address the research questions.

In line with the research goal, **MQR1** addresses the challenges posed by the complexity of object-oriented programs' features on change impact analysis and the imprecision associated with the static change impact techniques. This was achieved via the use of intermediate source code representation of object-oriented programs using complex software networks which can be used to quantify both change impact and program complexity. To facilitate impact analysis, the intermediate representation was transformed into adjacency matrixes which were used alongside the impact model based on change and dependencies types to precisely predict true impact sets. The impact analysis approach based on the impact diffusion range was designed to improve the precision of the static method. This helps software maintainers to clearly identify which components were truly affected by a change, and the impact set produced were small and practicable. In addition, the cost and efforts needed to perform changes were reduced considerably. We considered this approach very important in the perspective of teaching and learning of software maintenance at the undergraduate level due to its simplicity and ease of application when compared to existing change impact approaches of object-oriented programs. We also highlighted program comprehension strategies and models that these students would utilize to be effective and successful in object-oriented program maintenance.

In the same vein, the research question (**MRQ2**) was addressed using RQ2:1 and RQ2:2. Though MQR1 was basically to support change impact prediction in both small and large systems, **MRQ2** was only to support impact analysis in large-scale object-oriented programs. As we can see, object-oriented software is the mainstream in software development today, coupled with an inherent increase in size and complexity. In this case, activities such as inspection and testing could consume lots of time and resources to locate and fix faults. Therefore, it is important to have an approach that predicts which components will be faulty or fail in order to channel verification and validation activities early on such risky components before implementing a change. Thus, this was the rationale behind the support for impact analysis in large scale software systems as discussed in this thesis. We achieved this by conducting both comprehensive and systematic literature reviews of empirically validated processes and object-oriented product metrics respectively for use in the construction of a fault or failure prediction model. We found several software metrics we termed generic and which did not require further validation as predictors of class fault-proneness. These metrics were used to construct the failure prediction model in this thesis. Finally, we evaluated all the techniques discussed in this thesis and the results obtained were promising. We also developed a novel tool that assisted in the prediction of fault-prone classes which were impact set candidates before implementing a change.

8.2 Conclusions

In the light of the growing popularity, ubiquity and complexity of today's object-oriented software systems, it is therefore very important for organizations to maintain those systems effectively. In particular, understanding the change of large object-oriented software systems is a challenging problem as a result of their size today and the complex relationships among components introduced by features specific to the paradigm. This poses a great challenge to their maintenance, and change impact analysis in particular. In addition, the impact analysis methods used have not adequately addressed issues of imprecision and do not account for defects in the software components or support impact analysis from a beginner's point of view. Consequently, several approaches have been devised by researchers to maintain object-oriented systems via change impact analysis with the view to comprehend the complexity of object-oriented programs and with minimum impact set. However, these challenges have not been completely addressed.

To address the above challenges in this thesis, after reviewing the state-of-the-art in object-oriented features, software change impact analysis, and empirically validated software metrics in relation to fault-proneness, we combined the knowledge gained to devise an effective static change impact analysis approach. This approach supports the maintenance of object-oriented programs from the perspective of impact prediction and early failure with respect to fault-prone classes. In this case, to perform impact analysis, the maintainer has to construct an intermediate representation of the original software which is then transformed into adjacency matrixes. The matrixes in the perspective of class and member relations are then used alongside their impact diffusion rate to predict which components are truly affected by change. The impact diffusion range is based on the prevailing change type and the dependencies types. This approach therefore helps improve the precision of the method, reduces effort and time for changes. The approach is effective in the teaching and learning of software maintenance at the under graduate level to bolster the Software Engineering Education.

As software quality continues to be an irreplaceable key element in the success of any software organization, ensuring high quality in today's software development has become an increasingly complex and time-consuming activity. It is important to focus the available resources on the most critical parts of the system. Software metrics are used in this context to measure the quality and complexity of the software to ensure that the product satisfies functional and other quality requirements. Therefore, the early identification of the most critical software classes is important. In the perspective of change impact analysis, metrics are also important to measure and quantify change impact and risky changes in terms of faulty classes. In this thesis, we found different software metrics (process and product) that have been empirically validated as predictors of class

fault-proneness. Using these metrics on a binary logistic regression model will assist software maintainers in the prediction of which class will be fault-prone in the event of making a change. The results obtained from impact analysis will be beneficial to many activities such as regression testing, test prioritization and so on, which in turn are vital to project planning.

To validate the thesis, we conducted experiments and cases studies that evaluated the approaches. Based on the results obtained, the following findings were documented:

i. Intermediate source code representations of object-oriented programs that reveal their complex dependencies (semantically) and structures (syntactically) are effective for comprehending object-oriented program for onward maintenance.
ii. The key to object-oriented program change impact analysis is the knowledge of the different change types, the dependencies types and their impact diffusion range.
iii. To avoid costly changes in object-oriented software that could lead to software failure or increase the risk of failure in large software systems, faulty classes have to be predicted early in order to make effective and well-informed decisions before implementing actual changes on the system.

Other findings in this thesis in the perspective of the systematic literature review conducted and published in [18] are summarized as follows:

i. SLOC, CBO, RFC, WMC are metrics with a strong relationship with fault-proneness and are considered to be the best predictors of fault-proneness. LCOM is an indicator of fault-proneness in all the studies that considered CK+SLOC metrics, while DIT and NOC were found to be insignificant in most of the studies. From the results obtained, it shows that the best predictors of fault-proneness vary according to the class of applications as well as the application domain involved.
ii. Out of the twenty nine empirical studies of CK+SLOC metrics and fault-proneness of OO classes studied, six were students' projects while twenty three were not students' projects (mainly OSS and industrial applications).
iii. Only applications written in C++ and Java have so far been used for empirical studies involving the validation of OO metrics and fault-proneness.
iv. Prediction models built that utilize the variables were mostly based on logistic regression. Only a few machine learning and other techniques have been used. Therefore, logistic regression is the best statistical technique used for fault-proneness predictions.
v. The empirical studies circled around pre-release software products. Only one study was so far performed on a post-release (maintenance) product.

vi. Only a few replicated studies exist, though most studies reused datasets of previous studies.

Based on the above findings, we provide a number of recommendations:

1) To predict the fault-proneness of OO classes with some level of accuracy when CK+SLOC metrics are used, SLOC, CBO, RFC, WMC and LCOM should be used. In addition, logistic regression should be used as the predictive model since is the best model with high predictive power. In addition, other metrics such as DIT and NOC can only be considered based on their values measured in that particular software product. This is because, though they appear not to be regular class fault-proneness indicators, their significance or insignificance could be as a result of either the developers' experience or the inheritance strategy applied. Based on this, we cannot conclude on which of the metrics cannot be used as generic metrics for fault-proneness prediction of a class.
2) To ensure high quality software that is stable and maintainable, developers need to go for a low-coupled, highly cohesive design and controlled size and inheritance. In particular, strong focus should be placed on size and coupling as they appear to be strong and stable indicators of fault-proneness.
3) To assess the quality of OO software products either under development or maintenance, measures should not be based on the nature of the environment involved, but on steady indicators of design problems and impacts on external quality attributes.
4) More empirical studies should be performed on applications written in OO languages other than C++ or Java. In addition, more empirical studies should be performed in the academia and more replicated studies should be carried out in order to re-validate the metrics and keep them relevant.
5) Efforts should be geared towards post-release software products in order to confirm if models utilizing OO metrics can effectively predict class fault-proneness accurately or not.
6) During impact analysis of OO software systems, as a quality support, metrics should be used to assess first the software quality, in order to see where attention is most needed, before effecting changes.

In all, we suggest that developers and maintainers should use these metrics consistently to evaluate and then identify which OO classes require attention in order to channel resources to those classes or components that are likely to result in costly failures.

8.3 Research Limitation and Future Works

Impact analysis plays a critical role in the maintenance of software systems and at the same time, it is a very costly activity. Without change impact analysis, software maintainers would be

making changes in the "dark". In the light of this thesis, the approach we considered is restricted to changes and software change impact analysis in a software source code, object-oriented program in particular. Though many impact analyses exist in the perspective of requirements, design documents, test cases and so on, this work did not include them. Furthermore, fault or failure prediction is limited to only classes predicted to be affected by changes. Metrics validation is not included as only empirically validated metrics are used.

Also, there are other limitations that may affect the research and results. However, these were addressed in the applicable chapters in the form of validity threats, strengths and weaknesses, and limitations. We are also confident that, even if they did affect the results, their effects are insignificant to the research validation. The most important limitation in this research is that the static impact analysis technique is manual and has not been implemented to automatically build intermediate source code representations of the software that conducts change impact analysis to identify the true impact set. In addition, the evaluations we performed were on small scale applications developed by students. We therefore believe this will pose a limitation to a wider adoption of the proof of concept. Due to lack of current project data to evaluate the prediction model, we only used past project datasets to evaluate the model. Also the novel tool implemented will only predict if a class is faulty or not but cannot generate the model parameters such as the regression coefficient, R-square, maximum likelihood function and so on.

Based on these highlighted limitations, the future work of this thesis will be in the direction of implementing the static impact analysis technique in order to automatically analyze the original software, predict the impact of a change, extract software metrics and predict fault-prone classes. Though this will constitute a tedious work, much effort will be devoted to realize it in order to ensure that the time, cost, efforts and risks associated with software changes are eliminated or reduced while improving the quality of the product.

REFERENCES

[1] Lehman, M. and Belady, L. *Program Evolution: Processes of Software Change*. London Academic Press, 1985

[2] Bohner, S. A. "Extending software change impact analysis into COTS components" *Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop*, Greenbelt, USA, 2002, pp.175 -182.

[3] Grubb, P. and Takang, A.A. *Software Maintenance: Concepts and Practice*, 2nd ed. World Scientific Publishing Co. Pte. Ltd, Singapore, 2003

[4] Sommerville, I. *Software engineering* 9th ed. Addison-Wesley, New York, USA. 2011, pp.238-240

[5] Jönsson, P. and Lindvall, M. "Impact Analysis" *Engineering and Managing Software Requirements* Springer-Verlag Berlin Heidelberg, pp.117-142, 2005

[6] Bohner S, Arnold R. *Software Change Impact Analysis* IEEE Computer Society Press: Los Alamitos, CA, USA, 1996.

[7] Lee, M., Offutt, A.J. and. Alexander, R. T. "Algorithmic analysis of the impacts of changes to object-oriented software. *Proceedings of the Technology of Object-Oriented Languages and Systems* (TOOLS 00). Washington, DC, USA: IEEE Computer Society, pp. 61-70, 2000

[8] Jang, Y.K, Chae, H.S., Kwon,Y.R. and Bae, D.H. "Change Impact Analysis for A Class Hierarchy". *Proceedings of Asia Pacific Software Engineering Conference*, 1998, pp. 304 – 311

[9] Fenton, N., Ohlsson, N. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, Vol. 26 no. 8, pp.797-814, 2000

[10] Ohlsson, N., Alberg, H., 1996. "Predicting fault-prone software modules in telephone switches". *IEEE Transactions on Software Engineering* vol.22, no.12, pp.886-894, 1996

[11] Emam, K.E., Melo, W.L., Machado, J.C.: "The prediction of faulty classes using object-oriented design metrics". *Journal of Systems and Software*, No.56, pp. 63-75, 2001

[12] Sun, X. Li, B., Tao, C., Wen, W. and Zhang, S. "Change Impact Analysis Based on a Taxonomy of Change Types" *2010 IEEE Proceedings of 34th Annual Computer Software and Applications Conference* (COMPSAC 2010), 2010. pp. 373-382.

[13] Creswell, J.W.: *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 4th ed. SAGE Publications, London, United Kingdom. 2013

[14] Li, W., Henry, S. "Object-oriented metrics that predict maintainability". *Journal of Systems and Software,* vol.23, pp.111-122. 1993

[15] Zelkowitz, M. V. and Wallace, D. R. "Experimental models for validating technology". *Computer*, vol. 31, no. 5, pp.23-31, 1998

[16] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. *Experimentation in Software Engineering: An Introduction.* Norwell, MA, USA: Kluwer Academic Publishers, 2000

[17] Isong, B.E. and Ekabua, O.O. "Towards Improving Object-Oriented Software Maintenance during Change Impact Analysis", *Proceedings of the 12th International Conference on Software Engineering Research and Practice* (SERP'13) WORLDCOMP'13, CSREA Press, Las Vega, Nevada, USA, July 22-25.

[18] Isong, B.E. and Ekabua, O.O. "A Systematic Review of the Empirical Validation of Object-oriented Metrics towards Fault-proneness Prediction", *International Journal of Software Engineering and Knowledge Engineering* (IJSEKE) World Scientific Publishing Company December, 2013. Vol. 23, No.10, pp. 1513–1540

[19] Isong, B. and Ekabua, O. "Effective Representation of Object-oriented Program: The Key to Change Impact Analysis" *Proceedings of the International Conference on Software Engineering Research and Practice* (SERP'14) WORLDCOMP'14, CSREA Press, Las Vega, Nevada, USA, July 20-24. 2014

[20] Horowitz, E. Williamson, R. C. "SODOS: a software documentation support environment—its definition". *IEEE Transactions on Software Engineering*, vol.12. no. 8, pp.849–859, 1986.

[21] Pfleeger SL, Lawrence S. *Software Engineering: The Production of Quality Software.* Macmillan Publishing Co.: New York, USA, 1991.

[22] Turver, R.J. and Malcolm, M. "An Early Impact Analysis Technique for Software Maintenance," *Journal of Software Maintenance: Research and Practice*, Vol. 6, no. 1, January-February 1994, pp.35-52

[23] IEEE. The Institute of Electrical and Electronics Engineers, Incorporated. IEEE Standards 1219-1998: IEEE Standard for Software Maintenance.

[24] Lee, M. L.: "Change Impact Analysis of Object-oriented Software", A PhD dissertation, George Mason University, Fairfax, Virginia, USA. 1998

[25] Jarzabek, S. *Effective software maintenance and evolution: a reuse-based approach,* AuerBach Publications, Taylor & Francis Group NY, USA. pp. 28-33, 2007.

[26] Chapin, N. "Types of software evolution and software maintenance". *Journal of Software Maintenance and Evolution*, vol.13, no. 1, pp.1-30, 2001

[27] Kajko-Mattsson, M. "Motivating the Corrective Maintenance Maturity Model (CM3)", the *Seventh IEEE International Conference on Engineering of Complex Computer Systems*, 2001

[28] Erlikh, L., "Leveraging Legacy System Dollars for E-Business", (IEEE) IT Pro, May–June 2000, pp. 17–23

[29] Eastwood, A., "Firm Fires Shots at Legacy Systems", *Computing Canada*, vol.19, no. 2, 1993, pp.17

[30] Arthur L. J. *Software Evolution: The Software Maintenance Challenge*. New York: John Wiley & Sons, 1998

[31] Agarwal, B. B. and Gupta, T.M. *Software engineering and testing, Jones and Bartlett Publishers*, USA, pp.194 -195. 2010. ISBN 978-1-934015-55-1

[32] Chen, C.Y., She, C.W., and Tang, J. An Object-Based, Attribute-Oriented Approach for Software Change Impact Analysis, Proceedings of the 2007 IEEE, IEEM, 2007

[33] Badri, L., Badri, M. and Yves, S.D. "Supporting predictive change impact analysis: a control call graph based technique. *In Proceedings of Asia-Pacific Software Engineering Conference*, 2005

[34] Ekabua, O.O. and Adigun, M.O. "Reviewing Object Oriented Metrics for Aspect Oriented Paradigm", *Proceedings of International Conference on Software Engineering Research and Practice (SERP.07),* WorldComp.07, Las Vegas, Nevada, July, 2007

[35] Law, J., Rothermel, G., "Whole program path-based dynamic impact analysis", *The International Conference on Software Engineering*, 2003.

[36] Law, J. and Rothermel, G. "Incremental dynamic impact analysis for evolving software systems". *In Proceedings of International Symposium on Software Reliability Engineering*, 2003

[37] Apiwattanapong, T., Orso, A. and Harrold, M. J. Efficient and precise dynamic impact analysis using execute after sequences. *In Proceedings of International Conference on Software Engineering*, pp. 432 – 441, 2005.

[38] Pressman, R.S. *Software engineering: a practitioner's approach*, 5th ed. McGraw-Hill series in computer science, New York, USA. pp. I225-227, 200 SBN 0-07-365578-3

[39] Chowdhury, I. and Zulkernine, M. "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities". *Journal of Systems Architecture*, vol.57, pp.294–313, 2011

[40] Basili, V., Briand, L., & Melo, W. "A validation of object oriented design metrics as quality indicators". *IEEE Transactions on Software Engineering*, No.22, Vol. 10, pp.751–761, 1996.

[41] Chidamber. S. R and Kemerer. C. F, "A Metric Suite for Object-Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, no.6, pp.476-493, 1994.

[42] Harrison. R, Counsell. SJ, Nithi. RV, "An evaluation of the MOOD set of object-oriented software metrics". *IEEE Transactions on Software Engineering* , vol. 24, no.6, pp.491-496, 1998.

[43]. Daly. J, Brooks. A, Miller. J, Roper. M, Wood. M, "Evaluation inheritance depth on the maintainability of object-oriented software", *Empirical Software Engineering,* vol. 1, no. 2, pp.109-132, 1996.

[44] Fenton, N.E. and Neil, M. Software metrics: roadmap. In ICSE '00: *Proceedings of the Conference on The Future of Software Engineering*, ACM. New York, USA. pp. 357-370, 2000

[45] Briand, L., Daly, J., Porter, V., & Wust, J. "A comprehensive empirical validation of design measures for Object Oriented Systems". *Proceeding METRICS '98 Proceedings of the 5th International Symposium on Software Metrics IEEE Computer Society,* Washington, DC, USA, 1998

[46] Fenton, N. & Pfleeger, S. L. *Software metrics: A Rigorous and Practical Approach,* New York: Chapman and Hall, 1991

[47] Moreau, D. R. and Dominick, W. D. "A programming environment evaluation methodology for object-oriented systems: part I - the methodology," *Journal of Object-Oriented Programming,* vol. 3, pp.38- 52, May/Jun. 1990

[48] Kaur, K., Kaur, A. & Malhotra, R. "Alternative methods to rank the impact of object-oriented metrics in fault prediction modeling using neural networks". *World Academy of Science, Engineering and Technology*, vol.2, pp.877-882, 2008

[49] Shapiro, S.S. and Wilk, M.B "An Analysis of Variance Test for Normality," Biometrika, vol. 52, pp. 591-611, 1965.

[50] McCabe, T. J."A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp.308-320,1976.

[51] Abreu, F.B., Carapuca, R. "Object-oriented software engineering: measuring and controlling the development process". *Proceedings of the Fourth International Conference on Software Quality*, 1994.

[52] Lorenz, M., Kidd, J. Object-Oriented Software Metrics. Prentice-Hall, Englewood cliffs, New Jersey, USA. 1994

[53] Tang, M. H., Kao, M. H., & Chen, M. H. "An empirical study on object-oriented metrics". *In Proceedings of 6th IEEE International Symposium on Software Metrics.* pp.242–249, 1999

[54] Henderson-Sellers, B. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, Englewood cliffs, New Jessie, 1996

[55] Cartwright, M., Shepperd, M. "An empirical investigation of an object-oriented software system". *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 786-796, 2000

[56] Briand, L., Devanbu, P., Melo, W. "An investigation into coupling measures for C++". *In Proceedings of the 19th International Conference on Software Engineering*, 1997

[57] Li, L. and Offutt, A. J. "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software" P*roceedings of the 1996 International Conference on Software Maintenance,* ICSM 96, pages 171-184, 1996.

[58] Antoniol, G., Canfora, G. and De Lucia, A. "Estimating the size of changes for evolving object-oriented systems: a Case Study" *In Proceedings of the 6th International Software Metrics Symposium*, pp. 250-258, 1999

[59] Hattori, L. P. Guerrero, D., Figueiredo, J., Brunet, J.A. and Damasio, J. "On the precision and accuracy of impact analysis techniques," *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science* (ICIS'08). Washington, DC, USA: IEEE Computer Society, 2008, pp. 513–518.

[60] Ren, X. Shah, F. Tip, F., Ryder, B. G. and Chesley, O. Chianti: "A tool for change impact analysis of Java programs". *In Proceedings of Object Oriented Programming, Systems, Languages and Applications*, pp.432 – 448, 2004

[61] Tonella, P. "Using a concept lattice of decomposition slices for program understanding and impact analysis". *IEEE Transactions on Software Engineering*, 29(6):495 – 509, 2003.

[62] Sharafat, A.R. and Tahvildari, L."A Probabilistic Approach to Predict Changes in Object-Oriented Software Systems" *11th European Conference on Software Maintenance and Reengineering* (CSMR'07) March 2007, pp. 27-38

[63] Abdi, M. K., Lounis, H. and Sahraoui, H. "Analyzing change impact in object-oriented systems" *Proceedinds of 32nd Euromicro Conference on Software Engineering and Advanced*, 2006, pp.8

[64] Breech, B. Tegtmeyer, M. and Pollock, L. "Integrating Influence Mechanisms into Impact Analysis for Increased Precision" *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM06)*, 2006, pp.55-65

[65] Badri, L. Badri, M., St-Yves, D. "Supporting Predictive Change Impact Analysis: A Control Call Graph Based Technique," *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, IEEE Press, 2005, pp.167-175

[66] Oliveira, M. et al: "The Hybrid Technique for Object-Oriented Software Change Impact, Analysis" *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010),* IEEE Press, 2010, pp.252-255

[67] Chen, C. She, C. and Tang, J. "An object-based, attribute-oriented approach for software change impact analysis" *IEEE International Conference on Industrial Engineering and Engineering Management*, 2007, pp. 577-581

[68] Sun, X., Li, B., Tao,S.C., Chen,X. and Wen, W. "Using lattice of class and method dependence for change impact analysis of object oriented programs". *Proceedings of the 2011 ACM Symposium on Applied Computing*, pp.1439-1444, 2011

[69] Xiao-bo1, Z. Ying, J. and Hai-tao, W. "Method on Change Impact Analysis for Object-Oriented Program" *The 2011 Fourth International Conference on Intelligent Networks and Intelligent Systems*, 2011

[70] Hass, A. M. J. *Configuration Management Principles and Practice. Boston*, MA, USA: Addison-Wesley Professional, 2002

[71] Leon, A. *A Guide to Software Configuration Management*, Artech House incorporated, London, United Kingdom, 2000

[72] Koenemann, J., Robertson, S.P. "Expert problem solving strategies for program comprehension". *Conference on Human Factors in Computing Systems: Reaching Through Technology*. ACM Press, New York, USA, pp. 125–130, 1991

[73] Brooks, R. "Towards a theory of the comprehension of computer programs". *International Journal of Man-Machine Studies*, vol.18, pp.543-554, 1983

[74] Littman et al. "Mental models and software maintenance". *Empirical Studies of Programmers*, pp. 80-98, 1986

[75] Letovosky, S. Cognitive processes in program comprehension. *Empirical Studies of Programmers*, pp. 58-59. 1986

[76] Liu, J., Lu, J., He, K. and Li, B.: Characterizing the structural quality of general Complex software networks. *International Journal of Bifurcation and Chaos*, Vol. 18, No. 2 pp.605–613, 2008

[77] Pan, W.F., Li B, Ma Y.T. et al: "Measuring structural quality of object-oriented software via bug propagation analysis on weighted software networks". *Journal of Computer Science and Technology*, vol. 25, no. 6, pp.1202–1213, 2010

[78] Concas G, Marchesi M, Pinna S, Serra N. "Power-laws in a large object-oriented software system". *IEEE Transaction in Software Engineering*, vol.33, no. 10, pp.687-708, 2007.

[79] Vieira, M. R. E. "A Compositional Approach for Analyzing Dependencies in Component-Based Systems". *Information and Computer Science*. University of California, Irvine. Doctor of Philosophy, 2003

[80] Yu, P., Systa, T., & Muller, H. "Predicting fault proneness using Object-Oriented metrics: An industrial case study". *In Proceedings of Sixth European Conference on Software Maintenance and Reengineering*, Budapest, Hungary, pp.99–107, 2002

[81] Xu, J., Ho, D. and Capretz, L.F. "An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction". *Journal of Computer Science*, vol. 4, no.7, pp.571-577, 2008

[82] Subramanyam, R. and Krishnan, M.S. "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects". *IEEE Transactions on Software Engineering*. No.29, pp.297-310, 2003

[83] Janes, A. et al. "Identification of defect-prone classes in telecommunication software systems using design metrics". *International Journal of Information Sciences*, 2006

[84] Al-Dallal, J. and Briand, L.C.: "An object-oriented high-level design-based class cohesion metric". *Information & Software Technology*, No. 52, pp.1346-1361, 2010

[85] Succi, G., Pedrycz, W., Stefanovic, M., Miller, J. "Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics". *Journal of Systems and Software*, vol. 65, pp.1-12, 2003

[86] Malhotra, R., Kaur, A. and Singh, Y. "Empirical validation of object-oriented metrics for predicting FP at different severity levels using support vector machines". *International Journal System Assurrance Engineering Management*. No.1, vol. 3, pp. 269–281, 2010

[87] Shatnawi, R. and Li, W. "The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process". *The Journal of Systems and Software* no. 81, pp.1868–1882, 2008

[88] Emam, K.E., Benlarbi, S., Goel, N., Rai, S.N. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". *IEEE Transactions on Software Engineering.* No. 27, pp.630-650, 2001

[89] Briand, L.C., Wüst, J., Daly, J.W., Porter, D.V. "Exploring the relationships between design measures and software quality in object-oriented systems". *Journal of Systems and Software*, No. 51, pp. 245-273, 2000

[90] Olague, H.M., Etzkorn, L.H., Gholston, S., Quattlebaum, S. "Empirical Validation of Three Software Metrics Suites to Predict FP of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes". *IEEE Transactions on Software Engineering*, No.33, pp.402-419, 2007

[91] Zhou, Y., & Leung, H. "Empirical analysis of object oriented design metrics for predicting high severity faults". *IEEE Transactions on Software Engineering*, 32(10), pp. 771–784, 2006

[92] Aggarwal, K. K., Singh, Y., Kaur, A. and Malhotra, R. "Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on fault-proneness: A Replicated Case Study". *Software Process Improvement and Practice*, No.14, pp. 39–62, 2009

[93] Gyimóthy, T., Ferenc, R., Siket, I.: "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction". *IEEE Transactions on Software Engineering*, No.31, pp.897-910, 2005

[94] Singh, Y. Kaur, A. and Malhotra, R. "Empirical validation of object-oriented metrics for predicting FP models". *Software Quality Journal*, vol.18 pp. 3–35, 2010

[95] Briand, L.C., J. Wust and Lounis, H. "Replicated case studies for investigating quality factors in object-oriented designs". *Empirical Software Engineering*, No.6, pp. 11-58, 2001

[96] Olague, H.M., Etzkorn, L.H., Messimer, S.L. and Delugach, H.S. "An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study". *Journal of Software Maintenance,* No. 20, pp.171-197, 2008

[97] Rathore, S.S. and Gupta, A. "Validating the Effectiveness of Object-Oriented Metrics over Multiple Releases for Predicting fault proneness". *Proceedings of 19th Asia-Pacific Software Engineering Conference, IEEE*, pp.350-355, 2012

[98] English, M., Exton, C., Rigon, I. and Cleary, B. "Fault Detection and Prediction in an Open- Source Software Project", *In the 5th International Conference on Predictor Models in Software Engineering*, 2009

[99] Goel, B. and Singh, Y. "Empirical Investigation of Metrics for Fault Prediction on Object-Oriented Software". *Computer and Information Science*, pp. 255-265, 2008

[100] Shaik, A., Kurma, S.M., Riyazddin, M, Rao, S.V.A. Investigate the Result of Object Oriented Design Software Metrics on FP in Object Oriented Systems: A Case Study. *Journal of Emerging Trends in Computing and Information Sciences*, vol. 2 no.4, April 2011.

[101] Al-Dallal, J. "Transitive-based object-oriented lack-of-cohesion metric", *Procedia Computer Science*, pp. 1581-1587, 2011

[102] Zhou, Y., Xu, B. and Leung, H. "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems". *The Journal of Systems and Software* No. 83, pp. 660–674, 2010

[103] Pai, G.J., Dugan, J.B.: "Empirical Analysis of Software Fault Content and FP Using Bayesian Methods". IEEE Transactions on Software Engineering, No. 33, pp.675-686, 2007

[104] Johari, K. and Kaur, A. "Validation of Object Oriented Metrics Using Open Source Software System: An Empirical Study". *ACM SIGSOFT Software Engineering Note*, Vol. 37, No.1, pp.1, 2012.

[105] Myers, G., Badgett, T., Thomas, T., Sandler, C. *The Art of Software Testing*, second ed. John Wiley & Sons, Inc., Hoboken, New Jersey., 2004

[106] Kitchenham, B. and Charters, S. *Guidelines for performing Systematic Literature Reviews in Software Engineering*, Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 2007

[107] Arisholm, E. Briand, L.C. and Johannessen, E.B. "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models", *The Journal of Systems and Software*, vol.83, pp.2–17, 2010

[108] Moser, R, Pedrycz,W. Succi, G. "A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction". *ICSE '08. ACM/IEEE 30th International Conference on Software Engineering*, May 10–18, ACM. Leipzig, Germany, 2008

[109] Emad Shihab et al. "An Industrial Study on the Risk of Software Changes", *SIGSOFT'12/FSE-20, ACM*, Cary, North Carolina, USA, November 11–16, 2012

[110] Matsimoto, S. et al. "An analysis of Developer metrics for fault Prediction". *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, no. 8 2010

[111] Nagappan, N., Zellery, A., Zimmermannz, T., Herzigx, K. and Murph, B. "Change Bursts as Defect Predictors", *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010

[112] Bell, R.M. Ostrand,T.J and Weyuker, E.J. "Does Measuring Code Change Improve Fault Prediction?" Proceedings of the 2011th *International Conference on Predictive Models in Software Engineering (PROMISE '11)*, Banff, Canada September 20–21, 2011

[113] Krishnan, S. et al. "Predicting failure-proneness in an evolving software product line", *Information and Software Technology* vol. 55, pp.1479–1495, 2013

[114] Arisholm, E. and Briand, L.C. "Predicting Fault-prone Components in a Java Legacy System". *ISESE'06*, ACM. Rio de Janeiro, Brazil. September 21–22, 2006

[115] Graves, T. L., Karr, A. F., Marron, J. S., Siy, H. "Predicting fault incidence using software change history". *IEEE Transaction on Software Engineering*, Vol. 26, No. 7, July 2000

[116] Nagappan, N. and Ball, T. "Use of Relative Code Churn Measures to Predict System Defect Density" ICSE'05, ACM, St. Louis, MO, USA May 15-21, 2005

[117] Ostrand,T. J., Weyuker, E.J. and Bell, R.M. "Predicting the Location and Number of Faults in Large Software Systems". *IEEE Transaction on Software Engineering*, Vol. 31, No. 4, April, 2005, PP.340-355.

[118] Mockus, A. and Weiss, D.M. "Predicting risk of software changes", *Bell Labs Technical Journal*, 2000

[119] Weyuker, E. J. Ostrand, T.J. and Bell, R.M. "Using Developer Information as a Factor for Fault Prediction" T*hird International Workshop on Predictor Models in Software Engineering (PROMISE'07)* IEEE. 2007

[120] Illes-Seifert, T. and Paech, B. "Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs". Information and Software Technology, vol. 52, no. 5, pp.539–558, 2010

[121] Bell, R.M, Ostrand, T.J. and Weyuker, E.J. "Looking for bugs in all the right places" *Proceedings of the 2006 international symposium on Software testing and analysis*, ACM, New York, NY, USA,pp. 61–72, 2006.

[122] Weyuker, E. J., Ostrand, T. J. and Bell, R.M. "Adapting a fault prediction model to allow widespread usage". *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, ACM, New York, NY, USA, pp. 1–5, 2006.

[123] Menzies, T., Greenwald, J., Frank, A. "Data Mining Static Code Attributes to Learn Defect Predictors". *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp.1-12 2007.

[124] Nagappan, N., Ball, T., Zeller, A. "Mining Metrics to Predict Component Failures", *Proceedings of 28th International Conference on Software Engineering*, ICSE'06, Shanghai, China, May 20–28, 2006.

[125] Schr¨oter, A. Zimmermann, T and Zeller, A. "Predicting failure-prone components at design time", *In Proceedings of the 5th International Symposium on Empirical Software Engineering (ISESE 2006)*, ACM, Sept. 2006

[126] Sliwerski, J., Zimmermann, T and Zeller, A. "When do changes induce fixes?" *In Proceedings of International Workshop on Mining Software Repositories* (MSR), St. Louis, Missouri, U.S., May 2005.

[127] Cox, D. R., & Snell, E. J. *The analysis of binary data* (2nd ed.). London: Chapman and Hall, 1989

[128] Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K., Rosenberg, J. "Preliminary guidelines for empirical research in software engineering". IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 721–734, 2002

[129] Karahasanovic, A., Levine, A. K. and Thomas, R. "Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study". *The Journal of Systems and Software,* Vol. 80, pp. 1541–1559, 2007

[130] Soloway, E., Adelson, B. and Ehrlich, K. "Knowledge and processes in the comprehension of computer programs", *The Nature of Expertise*, pp.129–152, 1988.

[131] Mayrhauser, A.V. and Vans, A.M. Program comprehension during software maintenance and evolution. *Computer* vol. 28, no. 8, pp.44–55, 1995

[132] Metrics Data Program (MDP, 2006), http://sarpresults.ivv.nasa.gov/ViewResearch/107.jsp

[133] Hopkins, W. G. A new view of statistics. Sport Science. 2003, http://www.sportsci.org/resource/stats/