Chapter 1

Introduction and Literature Survey

1.1 Introduction

Measuring structural design properties of a software system, such as coupling, cohesion or complexity, is a promising approach towards early quality assessments. There are several metrics proposed in the literature to capture the quality of design and code. Most of these metrics are based on plausible assumptions but one key question is to determine whether they are actually useful, significant predictors of any relevant, external quality attributes such as fault-proneness, effort, productivity etc. We also need to explain how much they are applied in practice, whether they lead to effective models in a specific application context. The prediction models can be used by the
organizations during early phases of software development e.g. a model for prediction of software fault proneness allows software organizations to identify faulty classes and thus better focus on testing activities. These models provide a subset of metrics, which can be used by organizations among the large set of available metrics. The quality of the software can be assessed in the earlier stages of software development by computing the values of metrics found in the predicted model. These quality models can be used for system under development or maintenance. The work described in this thesis focuses on the use of Object-Oriented (OO) metrics in predicting the software quality attributes. This will help to identify which metrics are the important drivers of the quality attributes. The models constructed to predict quality attributes such as faulty classes may help in focusing testing and inspection resources on the fault prone parts of the design and code in a cost effective manner. The remainder of the chapter is devoted to a brief introduction of the field and disciplines related to this dissertation.

1.2 Software Metrics

The major research in the design phase of software development life cycle is the development of design metrics [91]. From the earliest days of software engineering there has been wide argument of the need of metrics for measuring software processes and products inorder to give clues to weak areas in the software development process. There is a great deal of literature in
this area that discusses the relevance and importance of measurement and its contribution to improved quality and productivity. Software metrics are initiated with the belief that they will improve software engineering, estimation, and management practices. The rationale arises from the notion that "you cannot control what you cannot measure" [44]. Software measures can help address the most critical issues in software development and provide support for planning, predicting, controlling, and evaluating the quality of both software products and processes. Software metrics can be defined as "The continuous application of measurement based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products" [63]. Lots of metrics have been proposed in last decade in the direction of design measurement of procedure-oriented and OO software.

The first and simplest measure of size has been the number of Lines Of Code (LOC). But LOC measure is too debatable. Counting of comment lines, data declarations, blank lines etc. may or may not be included in this metric [39]. Another measure proposed in the literature for the size was by Halstead, which is named as Science Measures [71]. The size measures were dependent on count of number of operators. Another significant measure of size was proposed by Albrecht [12] based on the count of function points, which measured the functionality of the software from users points of view. Most widely used and debated complexity measure was proposed by McCabe
This measure was based on control flow graph representation of the source-code and was used to provide a quantification of testing difficulty and hence an indication of reliability and maintainability. An in-depth study of 18 different categories of software complexity metrics was provided by Zuse [149], where he tried to give basic definition for metrics in each category.

Two main characteristics of any design are cohesion and coupling. Cohesion is an attribute of a software unit or module that refers to the relatedness of module components [145]. The most desirable form of cohesion is functional cohesion. In order to control and improve cohesion, a measure was needed which could measure the cohesion in the early phases of software such as design phase. Concept of coupling was first time defined by Steve et al. as the measure of the strength of association established by a connection of one module to another [133]. Myers defined six distinct levels of coupling to measure interdependence among modules [110].

One primitive and early measure of cohesion as well as coupling was suggested in the form of Information flow metrics [79]. Two new terms fan-in and fan-out were defined to measure the amount of information flow, and their multiplication was used to define the information flow metrics [79]. But several problems were identified with this information flow metrics [122], and then this metric was further extended to make it more sophisticated. A model of information flow was proposed [85], which was based on a modified definition of fan-in and fan-out.

The software organizations are focusing on software process improvement.
This demand led to new/improved approaches in software development area, with perhaps the promising being OO approach. The earlier software metrics (Halstead, McCabe, LOCs) were aimed at procedural-oriented languages. The OO paradigm includes new concepts (see Section 1.4). Therefore a number of OO metrics have also been proposed in literature in last two decades.

1.3 Software Quality Attributes

Software quality is not easy to define. Software quality is the degree to which software possesses a desired combination of attributes (example, reliability, maintainability, usability) [82]. As an attribute of an item, quality refers to a measurable characteristic. McCall [106], Boehm [22] and the International organization for Standardization (ISO) [86] provided a conceptual framework of quality attributes. McCall model of software quality was introduced in 1977. The model divides the quality attributes into two levels. Higher level quality attributes are known as quality factors. The second level of attributes are named as quality criteria. In 1978, Boehm introduced the quality model consisting of three levels of quality attributes. These levels are primary uses, intermediate constructs and primitive constructs. Intermediate constructs are similar to McCall’s quality factors and primitive constructs are similar to McCall’s quality criteria. However, this model includes hardware performance characteristics that were missing in McCall’s model [117].
In ISO 9126 hierarchical model each characteristic is related only to one attribute.

Quality has many attributes and some are related to each other. There are four attributes domains. These are usually the ones most entrusted by the customer [3]:

- Reliability: measures the extent to which a software performs its intended functions without any failure.
- Usability: is the degree of effort required to learn, operate and understand the functions of the software.
- Maintainability: is defined as the effort required to locate and fix an error through the maintenance phase.
- Adaptability is the degree to which a software is adaptable to new technologies and platforms.

1.4 Object Oriented Paradigm

Features of OO paradigm (programming languages, tools, methods and processes) provides support for many quality attributes [91]. The key concepts of OO paradigm are: classes, objects, attributes, methods, modularity, encapsulation, inheritance and polymorphism. An object is made up of three basic components: an identity, a state, and a behavior [23]. The identity distinguishes two objects with same state and behavior. The state of the
Object Oriented Paradigm

Object represents the different possible internal conditions that the object may experience during its lifetime. The behavior of the object is the way the object will respond to a set of received messages.

A class is a template consisting of number of attributes and methods. Every object is the instance of a class. The attributes in a class define the possible states in which an instance of that class may be. The behavior of an object depends on the class methods and the state of the object as methods may respond differently to input messages depending on the current state. Attributes and methods are said to be encapsulated into a single entity. Encapsulation and data hiding are key features of OO languages. The main advantage of encapsulation is that the values of attributes remain private, unless the methods are written to pass that information outside of the object. The internal working of each object is decoupled from the other parts of the software thus achieving modularity. Once a class has been written and tested, it can be distributed to other programmers for reuse in their own software. This is known as reusability. The objects can be maintained separately leading to locating and fixing of errors easier. This is called maintainability.

An OO software may be viewed as a set of objects that are related and exchange messages. An OO message is made up of:

- Receiver
- List of arguments
The receiver is the object which receives the message. The optional list of arguments holds the values with which the receiver will interpret the message. There are four kinds of relationships between classes: association, aggregation, dependency and inheritance. The most powerful technique associated to OO methods is the inheritance relationship. If a class B is derived from class A. Class A is said to be a base (or super) class and class B is said to be a derived (or sub) class. A derived class inherits all the behavior of its base class and is allowed to add its own behavior. Polymorphism (another useful OO concept) describes multiple possible states for a single property. Polymorphism allows programs to be written based only on the abstract interfaces of the objects which will be manipulated. This means that future extension in the form of new types of objects is easy, if the new objects conform to the original interface. OO exception handling mechanism helps in building robust programs and reduces cost of failure.

When analyzing a problem from OO point of view, the first things identified are actors (external entity that communicate with the system) and use cases (different functionalities that a system should provide). Thus once the behavior of the system has been constructed, the process of identifying the classes starts. After identifying the classes, the attributes and methods to be included in the class are determined. The Unified Modeling Language (UML) is the standard used for representing all the steps in the OO life cycle. UML includes definition of concepts, graphical representation of elements, and guidelines to how to use the notations. Each class should be
highly cohesive and less coupled with other classes [99].

The OO paradigm define a different way of doing things. This has led to the development of different methodologies that especially fit for the OO paradigm. One of the most important methodology is the Rational Unified Process (RUP) [97].

1.5 Literature Survey

In this section, we first provide the current state of research and results on OO metrics. We then describe the most relevant empirical studies regarding quality prediction in OO software.

1.5.1 Object Oriented Metrics

The first significant OO design metrics suite was proposed by Chidamber and Kemerer [37] in 1991. Then came another paper by Chidamber and Kemerer defining and validating metrics suite for OO design [36]. This metrics suite has received the most widest attention in empirical studies. Chidamber and Kemerer defined a suite of six design metrics-Coupling Between Object classes (CBO), Response For a Class (RFC), Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number Of Children (NOC), and Lack of Cohesion in Methods (LCOM) [36]. They used an automated data collection tool in collecting empirical data of these metrics from two commercial systems developed in small talk and C++. 
Li and Henry i) investigated proposed Chidamber and Kemerer OO metrics ii) proposed some OO metrics iii) validated the metrics using the maintenance data collected from two commercial system [101]. The authors collected the maintenance data for three years and measured maintenance effort in term of the number of lines changed per class. Li and Henry proposed some metrics-Message Passing Coupling (MPC), Data Abstraction Coupling (DAC), Number of Methods (NOM), SIZE1, and SIZE2. Li proposed a metrics suite consisting of six metrics - Number of Ancestor Classes (NAC), Number of Local Methods (NLM), Class Method Complexity (CMC), Number of Descendent Classes (NDC), Coupling Through Abstract data type (CTA), and Coupling Through Message passing (CTM) [102]. All except NDC had been already empirically validated in a previous study conducted by Li and Henry [101] in predicting maintenance effort. In 1996, Henderson-sellers [78] redefined the LCOM metric given by [36].

Another set of metrics for OO design consist of Operation Complexity, Operation Argument Complexity, Attribute Complexity, Class Coupling, Class Hierarchy, Cohesion, and Reuse. These metrics were empirically validated by using statistical regression analysis based data collected from two small projects and the data collected from the judgment of expert designers to the complexity of the design [34].

The approach by Bieman and Kang [20] to measure cohesion was based on that of Chidamber and Kemerer [36]. They proposed two cohesion measures: Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC). Lee
et al. acknowledged the need to differentiate between inheritance-based and non inheritance-based coupling by proposing the corresponding measures: Non Inheritance information flow-based coupling (NIH-ICP), Information flow-based inheritance coupling (IH-ICP) [100]. Information flow-based coupling (ICP) metric was the sum of NIH-ICP and IH-ICP metrics. Lee et al. emphasized that their ICP metrics based on method invocations, take polymorphism into account. They also proposed a cohesion metric based on information flow method invocations within a class known as Information flow-based CoHesion (ICH) [100].

F.B.E Abreu proposed another metrics suite consisting of six system level metrics known as MOOD (Metrics for Object-Oriented Design). The proposed metrics were Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (PF), and Coupling Factor (CF) [2]. The MOOD metrics were investigated by Harrison et al. [72]. They discussed each metric from measurement theory view point taking into account the OO concepts: encapsulation, inheritance, coupling, and polymorphism. Empirical data collected from three systems was used to analyze the MOOD metrics. The results showed that the metrics could be used to provide an over all assessment of software, but the authors stressed upon the need of more empirical studies, before the results could be generalized [72]. Lorenz and Kidd proposed a set of metrics grouped into four categories size, inheritance, internals, and externals [103]. Size oriented metrics for OO class focused on
counts of attributes and operation of an individual class and average values for OO software as a whole. Inheritance based metrics concentrated on the manner in which operations were reused through class hierarchy. Metrics for class internals were oriented towards cohesion, while external metrics were used to examine coupling and reuse [103]. Neal did a study for validation of OO software metrics and found that many of the proposed metrics couldn’t be considered valid measures of the dimension they claim to measure [113]. He defined a model based on measurement theory for validation of OO metrics and proposed a new suite of ten metrics - Potential Method Inherited, Proportion of Method Inherited by a Subclass, Density of Methodological Cohesiveness, Messages And Argument, Density of Abstract Classes, Proportion of Overriding Methods in a subclass, Unnecessary Coupling through Global Usage, Degree of Coupling Between class Objects, number of Private Instance Methods, and Strings of Message Links.

Abreu et al. provided a new framework for classifying metrics named a TAPROOT (TAXonomy PReis for Object Oriented meTrics) [1]. This framework classified metrics along two independent vectors: category and granularity. Six categories of OO metrics were defined: design, size, complexity, reuse, productivity, quality, and generic approach. They also proposed 3 granularity levels: system, classes, and methods. But no empirical/theoretical base for the metrics was provided [1]. Effectiveness of frameworks was studied and the authors found that the frameworks do not always deliver the concept of flexibility and reusability [66]. Then they developed a conceptual model for
the framework and a set of guidelines to build OO frameworks. The focus of guidelines was on improving the flexibility, reusability, and usability of frameworks [66]. In 2003, the usefulness of OO framework to domain specific business application has been studied from viewpoint of saving cost and improving quality of the software in a company [60]. Two case studies were conducted, in each of which four kinds of applications were developed. Each of the application was developed in two ways: using conventional module-based reuse and using framework-based reuse [60]. An automated tool was used to measure the reusability and its effect on quality by [54]. The results showed that reusability can be useful in improving the productivity and quality. But at the same time, excessive usage of frameworks for reusability has been reported to increase over all complexity.

Number of coupling metrics for OO software has been proposed by Briand et al. [25]. These metrics takes into account the different OO design mechanisms provided by the C++ language: friendship, classes, specialization, and aggregation. An unified framework for measuring cohesion and coupling was given by Briand et al. [27, 28]. They provided a mechanism for comparing metrics and their potential use, integrated existing metrics which examine the same concepts in different ways, and facilitated more rigorous decision making regarding the definition of new metrics and the selection of existing metrics for a specific goal of measurement. Benlarbi and Melo defined a suite of polymorphism metrics in 1999 [19] based on two kinds of polymorphic behaviors: compile-time polymorphism and run-time poly-
morphism. The metric suite includes a set of five metrics: Overloading in stand-alone classes (OVO), Static Polymorphism in Ancestors (SPA), Static Polymorphism in Descendants (SPD), Dynamic Polymorphism in Ancestors (DPA), and Dynamic Polymorphism in Descendants (DPD). Tang et al. also proposed a set new metrics that could serve as an indicator of how strongly OO a program is [136].

1.5.2 Existing Empirical Studies on Quality Models for Object-Oriented Software

A large portion of empirical research has been involved with the development and evaluation of the quality models for OO software. The immediate goal of this research was to relate structural attribute measures intended to quantify important concepts of OO software, such as

- encapsulation,
- coupling,
- cohesion,
- inheritance and
- polymorphism

to external quality attributes such as

- fault proneness,
- maintainability,
- testing effort,
- rework effort,
- reusability, and
- development effort.

The main motivation was to be able to assess quality of software in earlier phases of the software life cycle and to be able to use structural attribute measures for predicting external attribute measures. This would greatly facilitate technology assessment and comparisons, e.g., in studies such as [121].

As discussed in previous section, several measures for OO designs have been proposed (Li & Henry [101], Chidamber & Kemerer [36], Brito e Abreu [1], Briand et al. [25], Bieman & Kang [20]). The metrics suite proposed by Chidamber and Kemerer [36] are the most widely used and frequently referenced OO-design measures. These metrics have been validated in the literature in order to predict quality attributes.

Based on a study of eight medium-sized systems, developed by students Basili, Briand, and Melo [16] found that except LCOM all of the Chidamber and Kemerer metrics [36] were associated with fault proneness. In [16], it was concluded that Chidamber and Kemerer metrics proved to be better predictors than the traditional code metrics which can only be collected at
later phases of the software life cycle. In [43], an experiment was conducted to evaluate the effects of inheritance depth on the maintainability of OO software. Results suggested that systems with approximately three levels of inheritance depth may result in reduced time required to perform maintenance tasks by 20% compared to no use of inheritance. However, results from the same study indicated decreased maintainability at five levels of inheritance depth.

Chidamber & Kemerer provided an exploratory analyses of empirical data relating the metrics to productivity, rework effort, and design effort on three commercial OO systems [38]. In each of the three systems, the CBO and LCOM were shown as being significant independent variables. Harrison et al. evaluated MOOD metrics in 1998 [72]. They concluded that the MOOD metrics can be used at system level by the managers, providing an overall assessment of the system. However, they also stressed on the need of more number of empirical validations at the system level to establish relationships between metrics and quality attributes. In another study, Harrison et al. found the effect of inheritance on the maintainability of OO software. They concluded that the systems without inheritance were easier to modify as compared to the systems with three or five levels of inheritance. Sherif et al. analyzed two OO software project developed at jet propulsion lab - micro generic controller, and sequence generator. These two projects were analyzed and compared using three Chidamber and Kemerer metrics- WMC, DIT, and NOC. The empirical results collected for these metrics were used to get
insight into the complexity of two projects [123]. Wilkie et al. collected the data from 114 class sample for Chidamber and Kemerer metrics and noted some observations [143]: i) Cyclomatic Complexity can be used to measure the complexity of the class ii) WMC and DIT can be used to predict faulty classes iii) CBO had no effect on fault-proneness, but had an impact on maintenance effort iv) DIT and NOC combination gives a more thorough assessment of inheritance characteristic of a class.

Benlarbi and Melo [19] concluded that polymorphism might increase the fault proneness of OO software. Binkley and Schach [21] collected maintenance data from four development projects written in COBOL, C, C++ and Java, respectively. The results suggested that, “a significant impediment to maintenance is the level of interaction (or coupling) between modules”. Modules with low coupling were subjected to less maintenance effort and had fewer run-time failures and fewer maintenance faults.

Tang et al. [136] analyzed Chidamber and Kemerer [36] OO metrics suite on three industrial applications developed in C++. They found none of the metrics examined to be significant except RFC and WMC. Cartwright & Sheppard described an empirical investigation into an industrial OO software comprising 133,000 lines of code developed in C++ language [35]. They found that there was a significant difference in the defect densities between those classes that participated in inheritance structures and those that did not. They were also able to construct useful prediction systems for size and number of defects based upon simple counts such as the number of states
and events per class, but they concluded that such models may have local significance. Glasberg et al. validated OO metrics using a commercial Java application [62]. They found DIT and export coupling metrics to be good measures of fault proneness. Emam et al. examined a large telecommunication application developed in C++ and found that class size i.e. SLOC has confounding effect of most OO metrics on faults [52]. Emam et al., also predicted fault-prone classes using OO design metrics [53]. In their metrics suite, they included inheritance metrics (NOC and DIT) and various coupling metrics. The subject system analyzed was a commercial Java application (a word processor). They use a multivariate logistic regression method for analyzing the results. The results by Emam et al. indicated that DIT and export coupling measures were strongly associated with fault-proneness.

Briand et al. have extracted 49 metrics to identify a suitable model for predicting fault proneness of classes [29]. The system under investigation was medium sized C++ software system developed by undergraduate/graduate students. The eight systems under study consisted of a total of 180 classes. They used the univariate and multivariate analysis to find individual and combined impact of OO metrics and fault proneness. The results showed all metrics except NOC (which was found related to fault proneness in an inverse manner) to be significant predictor of fault proneness. Another study by [30] used a commercial system consisting of 90 classes. It found DIT metric related to fault proneness in inverse manner and NOC metric to be insignificant predictor of fault proneness. Briand et al. found the relation-
ship between polymorphism, Chidamber & Kemerer, and some OO metrics with fault proneness using two commercial systems consisting of 144 and 68 classes, respectively [31]. The results showed that the model predicted using multivariate adaptive regression splines outperformed the logistic regression model.

Fioravanti and Nesi extracted over 200 different OO metrics to identify a suitable model for detecting fault-proneness of classes [58]. The system under investigation was a medium-size C++ software system, developed by students attending an undergraduate /graduate level course. Then they employed statistical methods to produce a hybrid model comprised of 12 metrics. Two models for fault proneness prediction were developed and validated. They came to the conclusion that only few of the metrics were relevant for identifying fault-prone classes.

Yu et al. chose eight metrics and they examined the relationship between these metrics and the fault-proneness. The subject system was the client side of a large network service management system developed by three professional software engineers [146]. It was written in Java, consisted of 123 classes and around 34,000 lines of code. First, they examined the correlation among the metrics and found four highly correlated subsets. Then, they used the univariate analysis to find out which metrics could detect faults and which could not. They found that metrics (CBOin, RFCin, and DIT) were not significant while the other metrics to be significant predictor but to different extent.
Gyimothy et al. [67] empirically validated Chidamber and Kemerer [36] metrics on open source software for fault prediction. They employed regression (linear and logistic regression) and machine learning methods (neural network and decision tree) for model prediction. The results indicated that NOC was not significant predictor of fault proneness but all the other metrics were found significant in Logistic Regression (LR) analysis.

Olague et al. validated OO metrics on versions of an open source agile software [114]. They found WMC, CBO, RFC, and LCOM metrics to be very significant, while DIT was found insignificant in two versions of the system. NOC metric was found to be insignificant in one version while less significant in other two versions. Zhou and Leung validated the NASA data set to predict fault proneness models with respect to two categories of faults: high and low [147]. Pai also used the same data set using a Bayesian approach to predict fault proneness models [115].

In summary, OO code and design metrics have been used to empirically assess how structural concepts of OO systems affect software quality attributes. However, in literature [29, 30, 73] it has been stressed that more similar and replicated empirical studies should be carried out in order to draw stronger conclusions.
1.6 Goals of the Thesis

A number of OO measures are available in the literature. Researchers are also working hard to investigate the properties of these measures to understand the effectiveness and applicability of such measures. Hence, we need to understand what these measures are really capturing, whether they are really different, and whether they are useful indicators of quality attributes of interest?. In this work we perform a comparative review of existing studies and a replicated study in order to predict faulty classes using OO metrics. This will build a body of evidence, and present commonalities and differences across various studies.

There exist empirical studies that relate OO metrics and quality attributes. But these studies are few and more studies on different data set are needed in order to generalize conclusions. Thus recognizing this need we conduct empirical studies using publicly available NASA dataset, open source software, commercial software and student data set. There are many external quality attributes, but we focus on mainly fault proneness, testing effort and maintenance effort.

We need models to predict quality attributes at different severity levels of faults that can be used in earlier phases of software development, so that accurate project planning and testing planning can be performed. Using NASA data set we construct the models at different severity levels of faults. There are many machine learning methods which are being applied in
both classification and regression problems in various disciplines. There is a need to evaluate and compare their performance with respect to statistical methods. In this work, we analyze and compare the performance of various statistical and machine learning methods in predicting software quality attributes. We also propose and validate a software quality metric that can be used to predict faulty classes.

Thus, in this thesis, we focus on following primary goals:

1. To select subset of metrics that are related to quality attributes such as fault proneness, testing effort and maintenance effort using publicly available NASA, open source, commercial, and student data sets.

2. To compare different model prediction methods to find their performance and applicability.

3. To evaluate the effectiveness of OO metrics in predicting high/medium/low severity levels of faults.

4. To analyze the relationship between OO metrics and fault proneness using open source software.

5. To evaluate and compare various machine learning algorithms on different data sets in order to gain insight that which machine learning algorithm is performing better.
1.7 Organization of the Thesis

The organization of the thesis is presented in this section. Chapter 2 concentrates on the research methodology followed in this thesis. The subsequent chapters 3-10 present the results of the empirical studies carried out in this work. In chapter 11, a software quality metric is proposed and validated. Chapter 12 presents the conclusions of the work carried. Chapter wise description of the work carried is as follows:

Chapter 2. This chapter provides a description of the OO metrics, quality attributes, methods selected, and empirical data collection used in this work. A description of the initial steps followed in the research process is presented. The methods used to find the individual and combined effect of metrics on quality attributes are described. The performance measures used to evaluate the performance of predicted models are summarized.

Chapter 3. This chapter of the thesis compares and reviews the existing quality models available for predicting fault proneness of OO software in order to reflect the relevance of quality models to industrial practices. The models are compared in terms of results of the univariate and multivariate analysis for predicting fault proneness of classes. The commonalities and differences across different studies have been evaluated.

Chapter 4. The purpose of this chapter is to explore relationships between the existing design metrics and probability of fault detection in classes. The study described here is a replication of an analogous study conducted
by Briand et al. A quality model for prediction of fault proneness of OO software has been proposed and validated. We used the data collected from Java applications for constructing the prediction model. The study is divided into five parts i) descriptive statistics for each metric is presented ii) Principal Component (P.C.) method is used to determine whether these metrics are independent or not iii) The univariate analysis is carried out to test the hypothesis iv) relationship between design metrics and size of the class is analyzed v) a prediction model is developed and tested.

Chapter 5. This chapter examines and compares the Logistic Regression (LR), Decision Tree (DT), Support Vector Machine (SVM) and Artificial Neural Network (ANN) methods. These methods are explored empirically to find the effect of OO metrics given by Chidamber and Kemerer on the fault proneness of OO system classes. Data collected from Java applications have been used in the study. The performance of the methods was compared by Receiver Operating Characteristic (ROC) analysis.

Chapter 6. The aim of this work is to find the relation of OO metrics with fault proneness at different severity levels of faults. For this purpose, different prediction models have been developed using regression and machine learning methods. We evaluate and compare the performance of these methods to find which method performs better at different severity levels of faults and empirically validate OO metrics given by Chidamber and Kemerer. The results of the empirical study are based on public domain NASA data set. The performance of the predicted models was evaluated using ROC
analysis.

Chapter 7. The open source software development helps in producing timely and low cost software quickly and inexpensively. Several companies are funding open source projects because they use these software in their own work. Hence, it is important to measure the characteristics (such as coupling, cohesion and inheritance) of open source software. In this chapter, we find relationship between OO design metrics given by Chidamber and Kemerer and the fault proneness. We use open source software JEdit developed in Java language. We validate the usefulness of OO metrics for fault prediction using statistical and machine learning methods.

Chapter 8. Random Forest (RF) algorithm has been successfully applied for solving regression and classification problems in many applications. This chapter evaluates the capability of RF algorithm and compares its performance with nine statistical and machine learning methods in predicting fault prone software classes using open source software. The results indicate that the prediction performance of Random Forest is generally better than the statistical and machine learning models.

Chapter 9. The J48 DT algorithm has been successfully applied for solving classification problems in many applications. This chapter evaluates the capability of the J48 algorithm and compares its performance with nine statistical and machine learning methods in predicting fault prone software classes using publicly available NASA data set.

Chapter 10. This chapter examines the application of the ANN for
prediction of testing and maintenance effort using OO metrics. Quality estimation includes estimating lines of code changed or added during the life cycle of software. The dependent variable in our study is testing effort and maintenance effort. The independent variables are OO metrics given by Chidamber and Kemerer.

Chapter 11. In this chapter, we develop a software quality metric and suggest ways in which software professional may use this metric for process improvement using Gene Expression Programming (GEP). Here, we explore the ability of OO metrics using defect data for open source software. We conclude that the GEP can be used in detecting fault prone classes. We also conclude that the proposed metric may be effectively used by software managers in predicting faulty classes in earlier phases of software development.

Chapter 12. The final chapter includes the conclusions of the research work discussed in the thesis and suggests few directions for further research.