Master Thesis
Software Engineering
March 2012

Empirical evaluation of defect identification indicators and defect prediction models

Qui Can Cuong Tran

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Qui Can Cuong Tran
Address: Kurt-Schumacher Strasse 34, App 108, Kaiserslautern, Germany, 67663
E-mail: longphithien@gmail.com

External advisors:
Fabian Zimmermann
Steffen Olbrich
Fraunhofer Institute for Experimental Software Engineering, in Kaiserslautern, Germany
Phone/Fax: +49 631 6800-2135/-92135

University advisor:
Wasif Afzal
School of Computing

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden

Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Context. Quality assurance plays a vital role in the software engineering development process. It can be considered as one of the activities, to observe the execution of software project to validate if it behaves as expected or not. Quality assurance activities contribute to the success of software project by reducing the risks of software's quality. Accurate planning, launching and controlling quality assurance activities on time can help to improve the performance of software projects.
However, quality assurance activities also consume time and cost. One of the reasons is that they may not focus on the potential defect-prone area. In some of the latest and more accurate findings, researchers suggested that quality assurance activities should focus on the scope that may have the potential of defect; and defect predictors should be used to support them in order to save time and cost. Many available models recommend that the project's history information be used as defect indicator to predict the number of defects in the software project.
Objectives. In this thesis, new models are defined to predict the number of defects in the classes of single software systems. In addition, the new models are built based on the combination of product metrics as defect predictors.
Methods. In the systematic review a number of article sources are used, including IEEE Xplore, ACM Digital Library, and Springer Link, in order to find the existing models related to the topic. In this context, open source projects are used as training sets to extract information about occurred defects and the system evolution. The training data is then used for the definition of the prediction models. Afterwards, the defined models are applied on other systems that provide test data, so information that was not used for the training of the models; to validate the accuracy and correctness of the models
Results. Two models are built. One model is built to predict the number of defects of one class. One model is built to predict whether one class contains bug or no bug..
Conclusions. The proposed models are the combination of product metrics as defect predictors that can be used either to predict the number of defects of one class or to predict if one class contains bugs or no bugs. This combination of product metrics as defect predictors can improve the accuracy of defect prediction and quality assurance activities; by giving hints on potential defect prone classes before defect search activities will be performed. Therefore, it can improve the software development and quality assurance in terms of time and cost

Keywords: defect prediction; defect indicators; defect prediction models; quality assurance.

CONTENTS

1 INTRODUCTION

1.1 Motivation

Quality assurance plays a vital role in the software engineering development process. Within this process, quality assurance activities are applied to observe the execution of a system or software application to validate if the system or software application behaves as expected [4]. In addition, software testing is supposed to validate the correctness of the software system and to identify potential problems of the system.

An effective testing can, for example, be evaluated by the amount of defects found, i.e. the more defects discovered, the more effective the testing activities are. For instance, testing activity A discovers 10 defects, whereas testing activity B finds 50 defects of the same module within a system. Thus, we can say testing B is more effective than testing A. However, the effort that is spent on quality assurance activities, which is essential for the proper development of large and complex software system, is often not sufficient. This is due to time constraints and other factors, thus presenting a tough challenge [5]. Hence, all quality assurance activities involved in improving the software development and management processes are important research areas.

One major issue of ineffective quality assurance activities is that test activities may not focus on potential defect-prone areas. In both industrial practice and empirical research in software engineering, defining learning-based oracles for predicting defect-prone modules in a software product is one of the typical applied approaches [6, 7, 8]. Based on this approach, defect-prone oracle models of a new software project are created to predict defect-prone modules by studying the history of defect tracking data of the projects within the company or of comparable projects of other companies [6, 9].

Most of the proposed prediction models are based on various predictors or indicators, for example, product metrics such as size or complexity. There are plenty of size and complexity metrics that can be used within the defect prediction models. Table 1 gives an overview of the most common ones [10, 11, 12, 13]:

Metric	Description
Weighted methods per class (WMC)	The values of WMC are equals to the number of methods in the measured class.
Number of children (NOC)	This metric simply measures the number of immediate descendants of the class.
Number of public class	This metric counts the number of classes in the measured scope.
McCabe's cyclomatic complexity (CC)	It measures the complexity of executable code within the procedures. Cyclomatic complexity is probably the most widely used complexity metrics in software engineering.
Lines of Code (LOC)	This is the oldest and most widely used way of size metric to count lines of code. Depending on what we want to count, there are several ways to count the lines of code, i.e., counting all lines but exclude empty lines or comments, etc.

Table 1: The common metrics

Also, there are other ways that are less effective and/or less well-known.

Moreover, it has also been examined that the majority faults of a software system typically contained within certain modules of the system [13, 14]; this is called the defect content of software. Accordingly, the timely testing activities performed to identify these modules will facilitate the testing resources and give appreciable improvement for the software development processes. One way to identify the defect content of a software system is to apply code metrics to identify conspicuous modules. Therefore, it is required to investigate the relationship between the indicators and defect content of the software in order to save effort, time and cost for testing activities and improve the product quality of the software.

1.2 Goals and objectives

The main goal of this thesis is to investigate and identify the correlations between various code metric-based indicators and the defect content of the software systems. Thereby, a combination of indicators is defined to identify the defect content of software systems, which will take the advantages from the existing indicators. In order to validate the defined indicators and their correlation to the defect content, appropriate test systems need to be identified on which the analysis will be performed.

The objectives of this research work are listed below:

- Study and analyse the current metric-based defect indicators
- Define new defect indicators
- Analyse appropriate systems for evaluation, e.g. open source systems
- Identify the correlation between defined indicator and defect content
- Develop a new model that can predict the defect content.

1.3 Research questions

The thesis work will focus on answering different research questions in order to achieve the main goal as stated above. The research questions are listed below:

RQ1. What are the problems related to currently suitable indicators?

Problems and trade-offs should be identified to determine if an influence on the result of defect indicators is in software projects.

RQ2. What are the improvement opportunities for these indicators?

Can we find ways to improve or combine existing defect indicators for improving defect prediction?

RQ3. What are the opportunities for proposing a new defect prediction model?

Based on existing prediction models, we want to identify potential drawbacks of them in order to propose a new model to solve these drawbacks.

In practice, we can use these prediction models to support the release management. For example, by predicting the number of defects of this release, we can improve the quality of the software project as well the preparation of the test support management.

1.4 Thesis Methodology

Industrial resources and theoretical knowledge that should be observed in order to find the answers for the problems either in academic or industry [3].

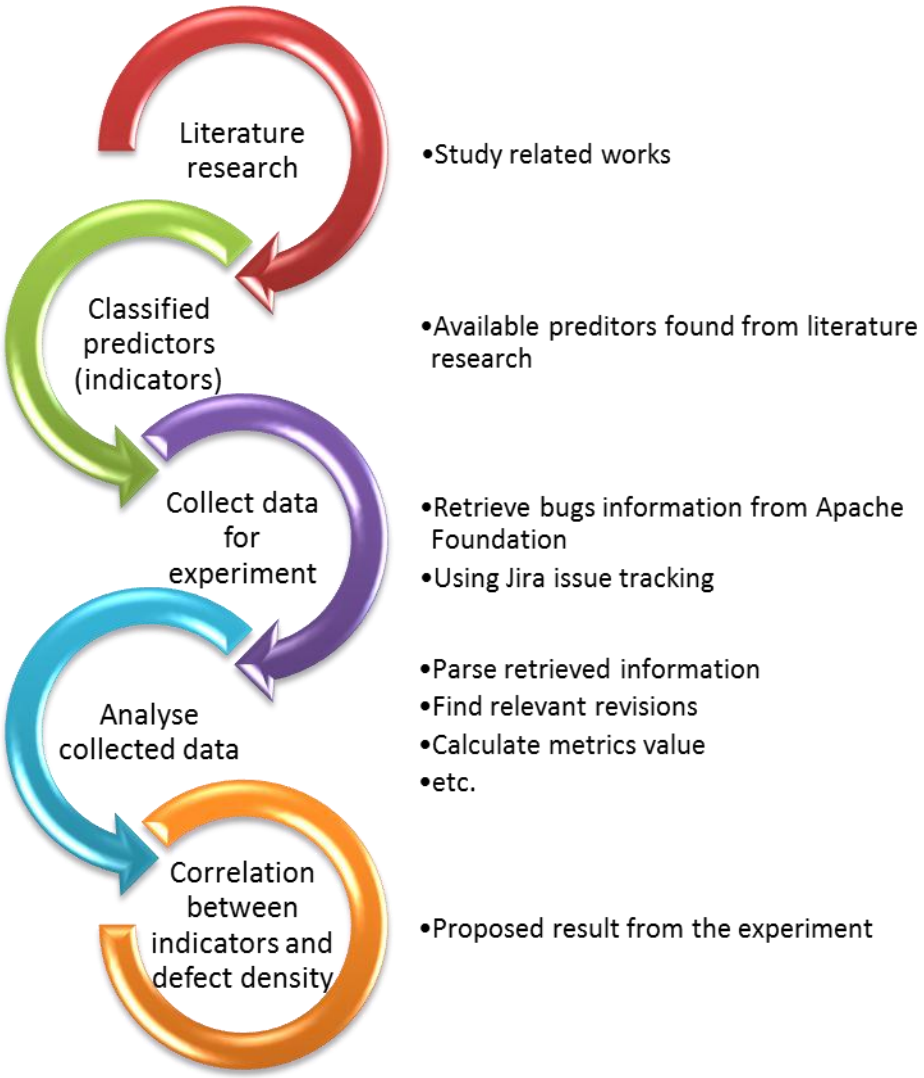

Figure 1: Research Methodologies

1.4.1 Mixed Method Approach

In this thesis work, the proposed research method will be the mixed method that is the combination between qualitative and quantitative research methods. The research work is defined in five phases, shown in Figure 1.

First phase – Literature research: as every research work, a systematic literature review is conducted to get all the currently suitable defect indicators for this work by finding all the relevant papers, articles, journals, references and related works.

Second phase – Classified predictors: As the result from systematic review, a list of classified predictors or indicators is defined to support for the experiment and making decisions on proposed model later.

Third phase – Collect data for experiment: Preparation for the experiment: appropriate open source project or software is chosen and identified to get the respective defect content as the data baseline for the experiment.

Fourth phase – Analyse collected data: Analyse the retrieved data from the third phase to find out any relationship between defect indicators and defect density of software project.

Fifth phase – Correlation between indicators and defect density: The results from the experiment will be taken into account to propose a new model for prediction and estimation the defect content.

1.4.2 Qualitative methods

For the systematic literature review, it is imperative to go comprehensively through articles, workshops, journals and conferences that have been published. The sources database consists of IEEE, ACM and SCOPUS. The material got from the literature review is used to find suitable solutions. The research questions RQ1 (i.e., What are the problems related to currently suitable indicators?) and RQ2 (i.e., What are the improvement opportunities for these indicators?) will be answered after performing the literature review about the defect indicators.

1.4.3 Quantitative methods

Industrial resources will be analyzed by performing the identification of the open source projects to find the respective defect content. The object of the experiment is Apache Software Foundation[1]. A project is chosen in order to retrieve bugs information and historical data of that project to support the experiment. The reason we selected an open source project from Apache Software Foundation is that this is a trustworthy software foundation; it has better feedbacks from users. Moreover, the chosen project has had a good history and is applied widely. More importantly, it is well-known in the software community.

After the data is collected, an analysis on it is performed to find any correlation between historical data and defect content of software project. This will be the basis sources to answer the research question RQ3 (i.e., What are the opportunities for proposing a new defect prediction model?).

Research question	Methodology
RQ1	Systematic literature review
RQ2	Systematic literature review
RQ3	Experiment/Industrial practices

Table 2: Research questions

1.5 Thesis outline

This section describes the structure of this thesis paper. Chapter 1 gives an introduction of quality assurance and to show its importance in software development processes. Additionally, the aims and objectives of this thesis will be discussed. Also, the thesis methodology is figured out in order to achieve the thesis's goals.

Chapter 2 discusses the background of the topic, i.e., definitions of defect and the different types of defects. Also, it describes existing defect prediction models and indicators that are used in both industry and research.

Sub-chapter 2.2 describes the result of literature review. It presents the related works, i.e., papers, articles, journals, etc. that are related to this topic. It also introduces

[1] The Software Apache Foundation http://projects.apache.org/projects/lucene_java.html (last visited on 15th, July, 2011)

some suggested approaches for defect prediction. There are two approaches: time-based approach and metrics-based approach. They will be described in more details in this sub-chapter.

Chapter 3 introduces and discusses the experiment. It describes in details which tools we used to collect data, which sources were used and how analysis data was conducted after data was collected.

Chapter 4 continues with the result we got from the experiment; it presents the correlation between defect indicators and defect content of software project. Two models are created from the collected data described in Chapter 3.

Chapter 5 describes the validation of the proposed models. They are validated with data from another open source project.

Chapter 6 discusses the threats to validity such as difficulties we have to deal with during the experiment and how we overcome them.

Finally, Chapter 7 summarizes the thesis and gives a conclusion about it.

2 FOUNDATION

2.1 Quality Assurance

"Today in some areas the survival of people depends on the correct function of software". [Prof. Dr. Liggesmeyer, UKL].

Software testing is an activity that has to be carried out during the software development life cycle. Software is developed by humans, and it usually contains flaws of humans. No one can ensure that the software is perfectly developed according to its intended design. Consequently, the development is not finished after the implementation phase; therefore, quality assurance activities are needed to check the system's correctness.

Quality assurance (QA) is the processes used to verify or determine whether products or services meet or exceed customer expectations or whether software or application behaves as expected from tested results [18]. Today, it plays an important role in software projects. It can help a software organization identify problems or errors of the product at early stage of the project or at every phase of the development processes. In this sense, in-time and efficient QA activities are required to ensure that they can get the crucial results of the software project quality.

QA processes are a set of activities that include a planned system of review procedures conducted by personnel who is not directly involved in the development or compilation processes [19]. These activities aim to check all of the customer's satisfaction and requirements and to make sure that the software behaves as designed. Moreover, QA activities can identify problems in order to reduce the risks of software's quality since this aspect obviously counts on the project's budget. Thus, accurate planning, launching and controlling of QA activities can contribute significantly to the general success of the software projects [22].

In addition, due to the reason that the technologies are developing and improving days by days and demand more complex systems, the modern software is increasing in size and complexity [1][4][16]. Consequently, software organization has to spend more money and effort on QA activities. There is a challenge for every software organization to manage the effort spending on these activities to make sure that their products are in good quality.

In some latest research, they recommend that test activities should focus on the potential defect-prone areas so that software organization could save the time and effort spent on QA activities. As a result, defect indicators are required for the expected number of defects that are found by QA activities. Indicators are used to predict the defect-prone of software project at early stage by estimating the number of latent defect in some areas that could contain bugs.

One of the main goals of QA processes is to initially find the defect density of the software product in order to prevent the software failure and increase the quality of product releases. The following section discusses about the defect and relevant area.

2.1.1 Definitions of defect

The term defect can be defined in various ways:

- Defect is any functionality that is wrong and it is said to be a defect by testers[2].
- If software misses some features or functions from what is in the requirement, then this is called as a defect[3].
- When any function does not work as it should work or those do not meet as per the requirement, then it is a defect[2].

Generally, the term "defect" is defined as "Statically existent cause of a failure, (i.e., a "bug"). Usually the consequence of an error made by the programmer" [Prof. Dr. Liggesmeyer, UKL].

Lyu [25] defined a defect's definition is when the discrepancy between failure and fault is not critical. Then the term "defect" is used to indicate it is either a "fault" (cause) or a "failure" (effect). Software defects can occur at any phase of the life cycle of the software development processes. They can be caught from the conceptual idea of the project until the end of the project. Defects are also known by other names such as errors or bugs. In this thesis, from now on, both terms are used synonymously.

Normally, a software project starts from scratch with a set of requirements from a customer or needs from the market. The gaps between requirements and product are bridged by some phases that take part in during analysis, design, code development and testing. This is called the life cycle of software development processes. All through the life cycle, if there is any further required changes to any phase, then that is considered as a defect. Hence, a defect can come from requirements inspection, analysis inspection, design inspection, code inspection, unit test or system test. All of these verification activities are conducted to make sure that the final product meets the intended requirement from the beginning.

Lyu [25] also described the defect-type attribute. The defect-type is something that occurred between missing and incorrect information. The defect types are described as follow:

- **Function defect** is the one that affects capability and functionality of the end product and requires a correct design change.
- **Assignment defect** refers to source code, e.g., error in control blocks or data structure.
- **Interface defect** indicates errors in communications between components or modules, etc.
- **Checking defect** is a failure when validating values or data such as loop conditions.
- **Timing/serialization defect** is errors related to shared and real-time resources.
- **Build/package/merge defect** indicates some errors in the library system or version control, etc.

In the scope of this thesis, according to the list of different types of issues defined in the JIRA issue tracking system – which is described in next chapter – the focus is on all defect types described above. They are related to the source code that causes some malfunctions in the software project, which result in defect report.

[2] Software bug http://en.wikipedia.org/wiki/Software_bug (last visited on Sep, 2011)
[3] What is a software defect http://www.dumpanalysis.org/blog/index.php/2008/01/08/what-is-a-software-defect/ (last visited on Sep, 2011)

2.1.2 Defect prediction, indicators and prediction models

Defect prediction can be seen as a necessary activity in the software development processes. Basically, defect prediction is used to assess an intermediate or final software product quality, estimate the number of (remaining) defects of the product or releases, or check whether all customer requirements and satisfaction are met or not [21]. It can be used to help the software organization know how good the product is, and also predict which part of the product could contain defects. This way they can focus on these and fix it. Also, it acts as a means of making better decisions for the release manager. Therefore, defect prediction can improve the software development processes in terms of time, effort and cost.

Generally, defect prediction activity intends to answer one of the following questions:

- Which metrics should be used to collect data through the early phase of software development processes as a good defect indicator?
- Which defect prediction models can be used for defect prediction activity?
- How good are these defect prediction models?
- How long does it take a software organization to adapt to these models for defect prediction?
- How much does it cost to apply these models?
- What kind of benefits do they bring to a software organization?

In the above questions, defect indicators are mentioned. However, what is a software defect indicator? Normally, in the source code of a software product, we can find different patterns that have a strong correlation with defect density and errors or faults in the source code that cause the software to malfunction. This is called the software defect indicator.

Defect indicators are metrics or a combination of metrics that provides insight into the software process, project or product itself. It can be used to indicate which part of source code of the software product that could contain defect so that software organization can focus on that part. Some examples of software defect indicator can be historical touch of files, historical data from the previous project, and unused variables in the source code or disable variables, etc.

Since we have all the required data for the prediction, we can form prediction models. Musa et al classified the models in five different attributes. They are time domain, category, type, class and family [27]. Currently, there are some models that suggested as appreciated models for defect prediction. These are listed as below [25]:

- Jelinski – Moranda de-eutrophication model
- Nonhomogeneous Poisson process model
- Schneidewind's model [26]
- Musa's basic execution time model
- Hyper exponential model

One of the popular models is the Software Reliability Growth Models (SRGM). Reliability is one of the probabilities of the system in that it will behave without errors at a specified time under a specified condition. SRGM is mostly used in software testing to identify the number of defects that QA activities is about to expose them in a certain time [28]. In the experiment at Tandem[4], Alan [29] observed that SRGM are used to give reasonable predictions of the number of (remaining) defects in the field. There are two types of software reliability models as described below:

[4] Tandem Computers, http://en.wikipedia.org/wiki/Tandem (last visited on August, 2011)

- First type tries to predict software reliability based on design parameters.
- Second type offers a prediction of software reliability based on test data.

The first type of software reliability models is usually well-known as defect density. They normally use some characteristics of source code such as lines of code, weight of classes, etc., to provide estimations about the numbers of defects or defect content of a software product. While the second type of software reliability models are called the software reliability growth models (SRGM). This type of models try to relate defect defection data to known functions statistically, e.g., exponential functions. The known function can be used to predict software's behaviour in the future in case that relationship is good enough [29].

Basically, these models are used to predict the number of defects, including found defects and remaining defects and defect density of software products. Hence, they can be used to tell the potential defect growth [16] and identify the body of knowledge for software quality measurement. This knowledge is required to measure quality throughout the life cycle of software development processes [23] in order to produce high-quality software. By learning this fact during the early stage, software organization can enhance the software quality; therefore, they can save the project's budget and reduce the risk as well as software errors. This is because poor software quality is the cause of software errors. Type of software errors are listed as below [24]:

- Code error
- Procedure error
- Documentation error
- Software data error

These errors can be reduced or prevented by using the indicators such as historical data from the previous projects and models to predict potential errors that software could have.

Most of the studies show the usage of indicators in order to predict the number of defects in software or the measurement based on the size and complexity of software [16][23][30][31]. Few studies really pay attention on defect content of software project [10][22].

For example, lines of code (LOC) is the oldest and most widely used size metric as a predictor. The more LOC a source code has, the more defects it may contain. It looks like a simple concept. However, LOC has some drawbacks that are listed below [47]:

- **Lack of accountability**: LOC is inaccurate and unfortunate to have to measure the productivity of a software with the outcome of development phase, which accounts only 30% to 35% the overall productivity of software product.
- **Adverse Impact on Estimation**: Consequently, from lack of accountability, any estimation is done based on LOC can adversely go wrong, in all possibilities.
- **Lack of Cohesion with Functionality**: Different developers may have different effort to develop functionality. For example, with the same functionality, skilled developer may create less code than another may.
- **Lack of Counting Standard**: There is no standard in the way to count lines of code or the definition of what a line of code is. The reason for this drawback that shows LOC is not completely accurate is whether comments are counted or not, whether data declarations are counted or not and whether a statement is over several lines.

Another example is the Cyclomatic Complexity (CC). It is a measure of the complexity of a program as a predictor. The more complexity the program is, the more defects it may contain. But, its drawback is that the same weight is placed on nested and non-nested loop; however, deeply nested condition structures are difficult to understand than non-nested ones.

The question is whether there is any relationship between defect indicators and defect density of software project. On the other hand, because any predictor has its advantages and drawbacks, it is conducive to study if there is any possibility to use a combination of these predictors in order to reduce the drawbacks. In summary, there is a need to study the correlation between indicators and defect content of a software project and the use of combination between predictors.

This might help to increase the software development processes in terms of time, quality and cost. Because with the information from this study, it enables the development organization where and which part of software development they should focus on and therefore, it can help them to save time and effort.

2.2 Defect prediction models

This sub chapter is the result of the first phase mentioned in sub-chapter 1.4 Thesis Methodology. A preliminary study on systematic review shows that there are many researchers who have worked with defect prediction indicators and defect prediction models. However, on most of their works, they have focused on the prediction indicators, e.g., how they can help to predict the defect density, or how effective they are in the software development. Common perspectives of the research in software engineering are identification of location of defects, clarification of causes and categorizing them.

2.2.1 Existing defect prediction models

There are not many systematic reviews or research on the correlation between the defect indicators and defect content of the software system. In addition, although previous studies have been reported on the same field, they do not really focus on the same issues that are mentioned in this thesis. This chapter gives an overview about the prior studies related to this thesis.

Jureczko et al. [1] conducted an analysis on newly collected repository with 92 versions of 38 proprietary open source and academic projects. The aim of this research work is to identify groups of software products with similar behaviours from the defect prediction perspective by performing clustering on the software projects. Hierarchical, k-means and Kohonen's neural network clustering were used to identify groups of relevant software projects. Therefore, a defect prediction model was created for each group to investigate relationship between that groups and defect content. However, a successful indicator was not mentioned in this paper. The results of this paper was the next step towards the defining methods of reuse defect prediction models by classifying groups of projects that the same prediction model may be used.

Illes-Seifert et al. [2] performed an empirical study to identify the relationship between the historical characteristics and software quality in open source products. The authors conducted an evaluation on nine open source Java projects across different versions. They used the defect detected count of a file as an indicator to predict its software quality and then the result of this measure was related to historical characteristics of that file.

This empirical study was performed on the open source java projects such as Apache Ant, Apache Formatting Objects Processor, Chemistry Development Kit, Freenet, Jetspeed2, Jmol, OSCache, Pentaho and TV-Browser. The authors extracted information contained in the versioning control systems of each project into history table in a database. Moreover, they also extracted the detected defects of the relevant project into a defect table in the same database. Afterwards, they performed the evaluation based on this database.

As a conclusion, they showed that historical data of software is a good indicator to predict its quality. In their research, based on historical data extracted from versioning control system, i.e., the number of defects in previous versions of a file, they estimated the number of expected defects for the future evolution. However, contrary to their expectation, there is no correlation between the defect count of a previous release of a file and its current defect count in most of their experiment's objects. Thus, they did not find one indicator that can persist across all the projects in an equivalent way.

Suffian et al. [14] proposed a new defect prediction model by using a combination of product metrics as indicators via the well-known methodology Six Sigma [15]. They performed an experiment on the software of the company where they were working for, named MIMOS Berhad. They used Design for Six Sigma methodology to build a defect prediction model which consists of 5 phases: Define – to identify the project goals and customer's requirement, Measure – to determine customer's needs and specification, Analyze – to analyze the process options to meet customer's needs, Design – to detail the process to meet customer's need and Verify – to confirm and prove the design performance to meet customer's needs.

Indicators were used in their work include: requirement error, design error, code and unit test (CUT) error, KLOC (thousands line of code) size, targeted total test cases to be executed, test plan error, test cases error, automation percentage, test effort in number of days and test execution productivity per staff day. In their result, a mathematical equation generated from the regression analysis based on these indicators has explained that defect prediction model could be constructed with the existence of identified factors. They concluded that, from the model equation, it discovered which strong factors contribute to the number of defects in testing phase.

Fenton et al. [16] conducted a research about common prediction model available in either industrial or academic field. They listed down the list of possible models used to predict the defect densities of software, e.g., prediction using size and complexity metrics, prediction using testing metrics, prediction using process quality data, prediction using multivariate approaches. Moreover, the authors also pointed out the weaknesses or problems of each prediction model. For instance, they showed either that size and complexity models assume that defect are a function of sire or program complexity causes the defects.

Although there are reports that correlation between complexity and defects, it is not obviously a straightforward one. As a result, they recommended using Bayesian Belief Networks (BBN) [17] for software defect prediction. The authors also showed the benefits of using BBN such as specification of complex relationship using conditional probability statements, which is easier to understand the chains of complex and seemingly contradictory reasoning via graphical format, etc.

The comparison of the related studies is shown in Table 3.

Paper	Year	Method	Objective	Aspect	DB-Range-No	Study type
Fenton [16]	1999	Systematic literature review	Defect prediction models		IEEE, ACM, cited by 527	Theoretical
Suffian [14]	2010	Metadata analysis	Defect prediction models		Scopus, ACM, IEEE	Empirical, Case studies
Jureczko [1]	2010	Systematic review + metadata analysis	Defect indicators		Scopus, ACM	Theoretical, Empirical, Case studies
Illes-Seifert [2]	2010	Systematic review + metadata analysis	Defect indicators		Scopus, ACM, IEEE	Theoretical, Empirical, Case studies

Table 3: Comparison of Related work

2.2.2 Suggested approaches for defect prediction

"Prediction is difficult, especially of the future", Niels Bohr.

This sub-chapter figures out and categorizes some predictors as a result from literature research. The goals of defect prediction activities aim to solve the following problems [16]:

- predict the numbers of found defects and the remaining defects of software project
- estimate system's reliability
- study the influence of software's design and testing activities on the numbers of defects and failure densities

Many studies have proposed a number of prediction models in both industry and academia. One study by Norman F. Schneidewind recommended two approaches for defect prediction activities [23]:

- First approach derives the knowledge requirements from a set of issues identified between time intervals, including artefacts of development process. It accounts for time which is called the time-based approach.
- Second approach focuses on specific issues; for example, cost and risk, context, models, product and process test evaluation, product and process quality prediction, etc. It also focuses on which measurement scales should be used to assess the product and process quality, which is called metric-based approach.

2.2.2.1 Time-based approach

Basically, this approach estimates the number of (remaining) defects of software project based on the numbers of defect found or defect occurrence times after the

project release in time intervals. Afterwards, the data is fitted on the software reliability growth model (SRGM)

The basic idea of SRGM is that they rely on a parameter that relates to the numbers of defect in a block of source code. Therefore, we can say how many remaining defects there are in the source code once we know both that the parameter and the numbers of defects found in the source code. This idea is described in the Figure 2. By predicting and understanding the remaining defects in source code, it can help software development organization to make quality decisions such as if the project is ready to deliver or not. It also helps the organization to prepare an appropriate support plan for customer after the software delivered in terms of time and cost [29].

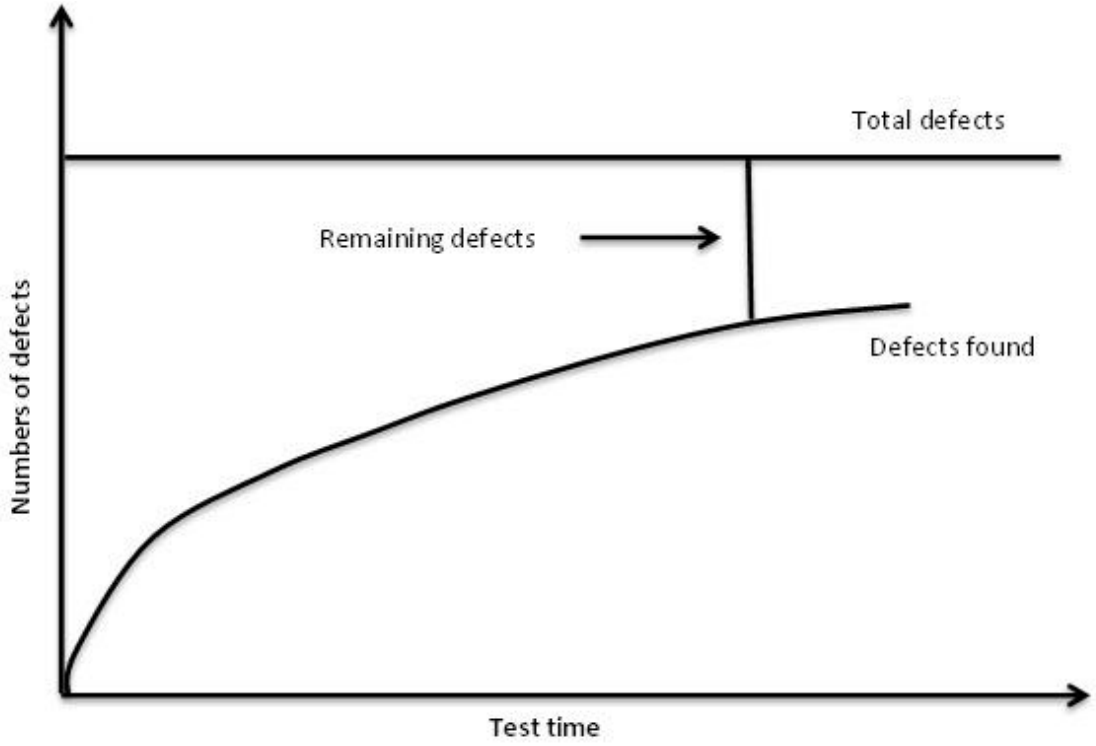

Figure 2: Remaining defects in software project [31]

Wahuydin et al [10] described in his paper using SRGM as a helpful model to sketch defect growth throughout the life cycle of projects in their experiment. First of all, they collected defect data of the project in their experiment by using strong predictors that have good correlation with defect growth between releases. Afterwards, they fitted these data on SRGM. Then, based on the models, they could assume about the productivity of the project such as the efforts that the development department should concentrate on to resolve the defects due to defect content of previous releases.

For instance, Figure 3 below is the SRGM for the project in Wahuydin's paper through the project's releases. In this model, the line is the estimated numbers of defects through 6 releases in 2 years, and circles are observed numbers of the defects related to each release within different time. According to this model, software organization could tell that from release C1.4 to C1.6, the numbers of defects discovered are a little bit higher than estimation. As a result, organization could make decision beyond the efforts that development department should spend to reduce the defect density and improve the productivity of the project.

The advantage of time-based approach is the accurate predictions. Since the estimations are derived from the actual defect occurrence data of the previous releases, this approach can offer predictions that are more accurate. At the same time, this advantage is also a disadvantage. Due to the reason that the necessary data for the estimation mostly come after the testing activities; therefore, the prediction using this approach is often too late to support the in-time decision making by software development organization such as release process, support plan process, etc.

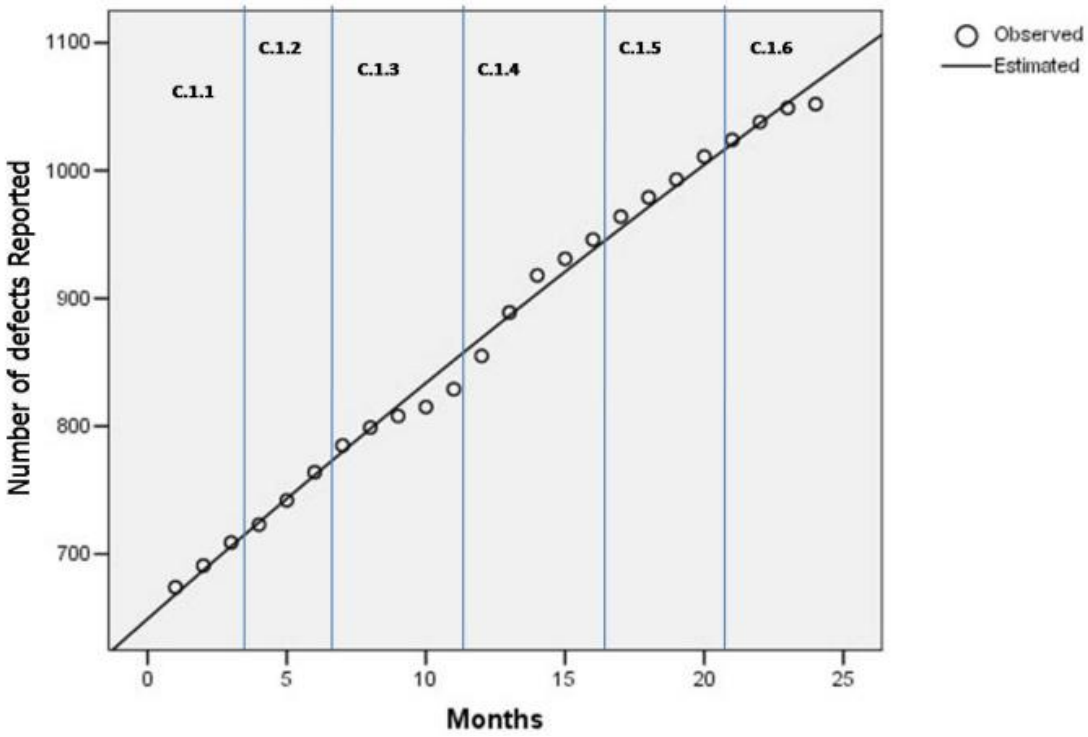

Figure 3: Software Reliability Growth Model [10]

Li et al [30] reported in their case study of OpenBSD that it is not possible to fit a SRGM to development defects by using time-based approach in order to predict the numbers of defects for OpenBSD. They discovered that defect-occurrence rates also increased commonly along with releases in term of time. Hence, it can be an impossibility to use SRGM such as Weibull[5] to predict the development defects. This issue opened the importance of metric-based approach to take into account.

2.2.2.2 Metrics-based approach

"You can't manage what you can't control, and you can't control what you don't measure.", Tom Demarco

Due to the reason that the defect prediction by using time-based approach is often too late to support the in-time decision making process of software organization such as release process and support plan. Therefore, metrics-based approach is suggested as an approach to cover the flaws of time-based approach.

Metrics-based approach is used to predict defect content of software system by using metrics obtained from historical project data before product release (it is called as predictor) to fit a predictive model [31].

[5] Weibull Distribution http://www.weibull.com/LifeDataWeb/the_weibull_distribution.htm (last visited on August 2011)

Metrics-based approach has four phases as showed in Figure 4: analyse, design, code and test/operate. The idea is that metrics-based approach can be applied for all phases of development processes; hence, it can be integrated in these processes. This approach can be used at any phase of the life cycle of the development process in order to improve the quality of software project.

This approach shows one of the aspects that can be considered for time. That is to say if predictions or measurements are performed at the early stage of life cycle, they can be less quantitative than the others obtained later [23]. Moreover, early measurements are based on static artefacts such as design document. On the other hand, later measurements are based on dynamic artefacts, such as code, and this is more quantitative than early measurements.

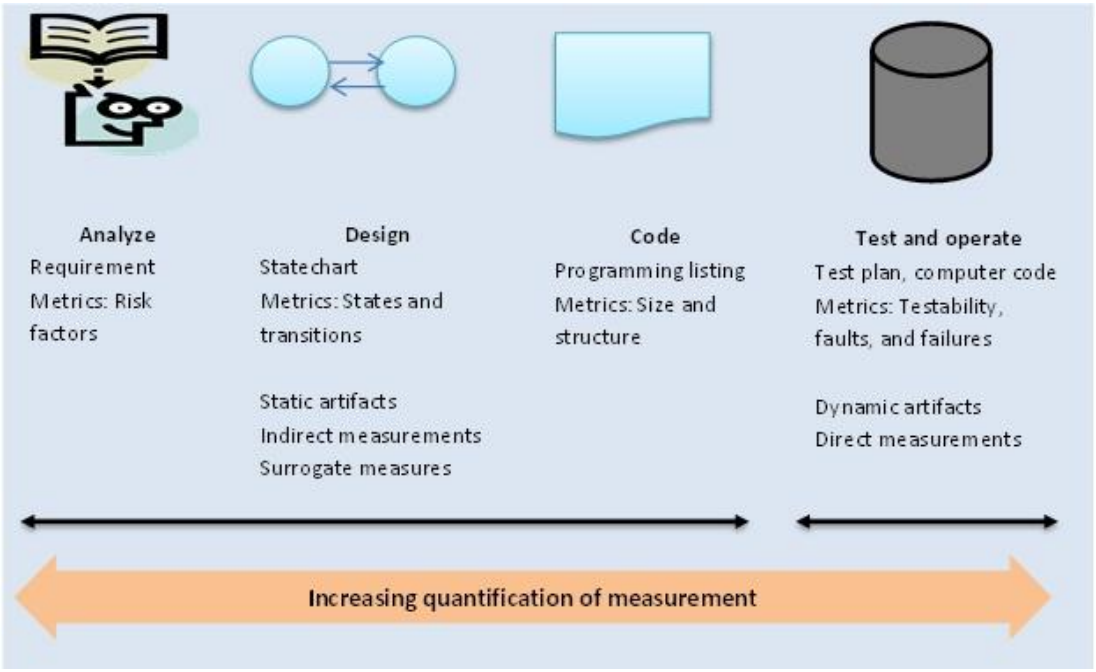

Figure 4: Life-cycle quality measurement [23]

In contrast to the time-based approach, metrics-based approach offers a better support of defect prediction in software project by using the historical information on predictors and current state of defect's density to fit a predictive model [10][31]. It provides defect prediction prior release so that software organization can have a better plan for release manager and support manager. Due to the reason that the metrics-based approach is based on historical information; therefore, it can focus on what area has the most potential defect-prone. This approach of the metrics-based models is more qualitative in accuracy.

Li et al [33] in their experiences at ABB Inc[6] categorized defect predictors into four groups: product metrics, project metrics, deployment and usage metrics, configuration metrics.

2.2.2.2.1 Product metrics

Metrics that used to measure the characteristics of any intermediate or final product of the software development process are called product metric [34]. They are

[6] ABB Inc http://www.abb.com/ (last visited on October 2011)

measurable ways to design and access the software product and are applied to the software project as a whole. Product metrics are used in software measurement, for example, size, complexity, coupling, etc [35].

- Size: lines of code (LOC)
- Complexity: McCabe Cyclomatic Complexity, Weighted Method for Class (WMC)
- Coupling: the number of the coupled classes

Purao et al [36] proposed that product metrics are good indicators for an organization's software engineers to have better understanding, designing and analyzing of the software project. They showed that product metrics can be integrated with some internal context provided by the development life cycle in order to improve the software measurement. For instance, in other similar studies, Denaro et al. [37] and Khoshgoftaar et al. [38], the authors have shown that product metrics have been important predictors and the most commonly used predictors in software measurement.

2.2.2.2.2 Project metrics

"If you can't measure it, you can't manage it", Peter Drucker

The above quote tells that if software organization cannot measure and manage a software project, they cannot improve the quality of software project. Project metrics are objectively measurable attributes of exciting project features. They are used to measure the attribute of development process and their activities. An example is the LOC/developer within software organization [39]. With these attributes, the organization can use these measurements of metrics to draw the information about the software quality.

The successful project metrics are derived from the main questions that concentrate on one particular aspect of the project. They require the ability to identify measures, which describe the essential parameters to control and improve the project. These questions and parameters can be categorized into some groups as suggested in PMBOK Guide [40]. For instance, project risk, project quality, project phase, etc. The categories are described in Table 4.

Weyuker [41] conducted an experiment to use developer information as a predictor for defect prediction. The authors derived three developer metrics: the number of developers, new developers who modified the file during the prior release, and the cumulative number of distinct developers who modified file during all releases through prior release. As the result, they reported that developer metrics were most strongly related to the number of defects at later releases. Hence, the authors suggested that these metrics are important predictors for defect prediction.

Name	Description
Cost	Will the project meet the budget?
Time	Will the project meet the schedule?
Scope	Will the project deliver planned scope? Does scope change from the beginning expectations?
Quality	Is the customer happy? (customer's satisfaction)
Risk	Are we effectively anticipating and managing risk events relevant to this project?

Table 4: Project metrics categories [39]

2.2.2.2.3 Deployment and usage metrics

The idea behind deployment and usage metrics is that they measure the attributes of the deployment context of the software system and usage patterns of software releases [Paul Luo Li]. There are not many studies about deployment and usage metrics [38]. The usage of these metrics can be counted on time since the first release, time to next release, number of ports on the customer installation or proportion of systems with a module installed [43].

2.2.2.2.4 Configuration metrics

Configuration metric is metric that is used to measure attributes of the software and hardware system or workstation that the software is installed and interact with the software product or release during the its operation [Paul Luo Li]. These metrics have been examined by few studies; for instance [42]. These are some examples of configuration metrics [43]: type of software application, system size of installation (small, medium or large), deployment operating system (Windows, Linux, etc.).

2.2.3 Conclusion on the related work

Due to the intended purposes of this thesis, we focus on the metrics-based approach and product metrics, which have deep insight on the potential of the prediction of defect content. Product metrics were chosen because of their benefits described below:

- Assist in the evaluation of the analysis and evaluation model
- Provide indication of procedural design complexity ad source code complexity
- Facilitate design of more effective testing

In practice, software engineers use product metrics to support and assess the software's quality and construction. In order words, product metrics provide software organizations with a basis to conduct analysis, design, coding and testing more objectively. In addition, the aim of this thesis is to propose a model to improve the QA activities. Thus, product metrics were selected as indicators to build a model.

As a result from the literature research, there are a number of product metrics that are used most in both industry and academia. They will be used to support for the making of the experiment's decisions. They are shown in Table 5, for more details, see Appendix A.

No	Metric name	Abbreviation
1	Number of Public Attributes	NOPA
2	Average Method Weight	AMW
3	Number of Methods	NOM
4	Access to Foreign Data – Class level	ATFDClass
5	Lines of Code – Class level	LOCClass
6	Number of Accessor Methods	NOAM
7	Weighted Of a Class	WOC
8	Weighted Method Count	WMC
9	Tight Class Cohesion	TCC
10	Maximum Nesting Level – included average and sum	MAXNESTING_Avg MAXNESTING_Sum
11	Number of Accessed Variables – included average and sum	NOAV_Avg NOAV_Sum
12	Changing Classes	CC_Avg CC_Sum
13	Cyclomatic Complexity	CYCLO_Avg CYCLO_Sum
14	Changing Methods	CM_Avg CM_Sum

Table 5: List of the product metrics

3 EXPERIMENT DESIGN

In this chapter, the processes of the experiment are described. It also introduces which tools and sources are conducted through this master thesis. This chapter is the result of the third phase (i.e., Collect data for experiment) and fourth phase (i.e., Analyze collected data) in the methodology mentioned above.

3.1 Tool selections

Throughout this experiment, the tool named "EvAn" which stands for EvolutionAnalyzer developed by Steffen M. Olbrich [20], was used. EvAn was developed for an automated detection of code smells and the collection of the impact data based on the historical development data of a software project. The analysis of impact data is based on the occurring changes and defects information. Afterwards, the analysis result is stored in different tables. The following section describes some important tables in EvAn database.

- **The defect_entities table** stores defect information, such as defect identification, defect's description, etc.
- **The revision_entities table** saves the information of all revisions, which were involved to fix defects, as well commit messages of these revisions.
- **The source_file_entities table** contains the information about which files were touched in one revision involved to fix defects.
- **The source_file_defect_entity_relation** table shows the relationship between source file(s) and defect. This relationship shows which source file(s) were touched to fix one defect.
- **The change_data_entities table** stores the detailed information which source file(s) were touched in which revision to fix defects and their repository.
- **The metric_results table** contains the information of metric results for classes and methods. These values were computed by several metrics.

The EvolutionAnalyzer is the conceptional structured in three independent parts: Version Control Extractor, Data Model Extractor and Analysis Subsystem. Figure 5 shows the conceptual overview of the whole system. The conceptual defined functionalities of EvAn can be summarized as follows [20]:

(a) gather data from source code repositories

(b) extract the meta-information of each revision (i.e., revision-date, author, change data, etc.)

(c) extract the metamodel of the system resident in the repository

(d) enrich this system meta-model with the code-metric results of the containing classes and methods

(e) map the change and defect data to the corresponding elements in the system meta-model

(f) create the links between classes and methods to their counterpart of a previous revision

(g) store this data in a local database

(h) perform further postprocessing and analysis.

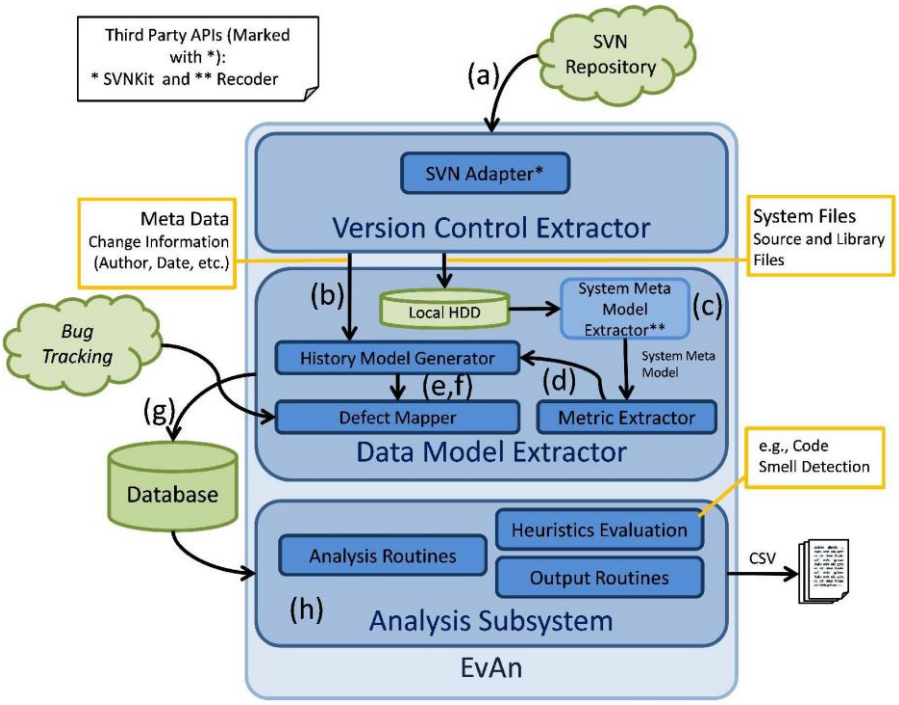

Figure 5: Schematic overview of the EvAn. [20]

In order to store these information described above, MySQL Workbench[7] was used as a database management for the EvAn tool. According to MySQL Workbench Help, it provides three main areas of functionality: SQL Development, Data Modelling and Server Administrator. In this concept, SQL development was used mostly for data processing.

3.2 Data collection

3.2.1 Selection of projects

The agreement was made that this evaluation would be done by performing on the Apache Software Foundation. At the first step, we selected the project "Apache Xerces for Java XML Parser". Apache Xerces[8] is a high performance, fully compliant validating XML parser, which is written in Java. Afterwards, we derived search strategy to retrieve rich and effective bug information as follow:

Source	**Apache Software Foundation**
Project	Apache Xerces for Java XML Parser
Bug/Issue Tracker	Jira
Issue type	Bug
Status	Resolved or Closed
Resolutions	Fixed

As a result, there were 615 bugs matching with the above query command. Subsequently, we used the EvAn tool to extract the revisions of Xerces2-J from its SVN repository. We retrieved 5402 relevant revisions from the SVN and performed matching between defect entities retrieved from Jira and revisions acquired from the SVN. However, unfortunately, only 225 out of 615 bugs fixed were matched to revisions. Hence, we found out that, for some reasons, when developers fixed bug and

[7] MySQL Workbench http://www.mysql.com/products/workbench (last visited on July 2011)
[8] ApacheXerces for Java XML Parser http://xerces.apache.org/xerces2-j/ (last visited on July 2011)

committed to SVN, they did not include bug information in commit message. For instance, there're a lot of commit messages missing the bug's KEY XERCESJ-xxxx, e.g. XERCESJ-1515. Therefore, it was unable to match these revisions to bug information.

Due to the lack of matching between bugs and revisions of the project Xerces2-J, we decided to choose another open source project, named "Apache Lucene Java". Apache Lucene[9] is a search engine library with high-performance and full-featured text, which is written entirely in Java. We applied the search strategy that carried out at the first step, with changed project, as follow:

Source	**Apache Software Foundation**
Project	Lucene – Java
Bug/Issue Tracker	Jira
Issue type	Bug
Status	Resolved or Closed
Resolutions	Fixed

As a result, we found 960 matching issues (last retrieved on Tuesday, 26.07.2011). After that, we used the tool EvAn to extract all revisions information from the SVN repository of project Apache Lucene Java. We retrieved 4531 revisions from the SVN.

3.2.2 Defect-prone declaration

We performed the matching process between these revisions and found bugs. After having the revisions table, we checked this table and found the last record of this table as shown in Figure 6:

ID	SYSTEM_ID	REVISION_NUMBER	DATE	AUTHOR	COMMIT_MESSAGE	META_MODEL_COMPLETE	CHANGE_DATA_COMPLETE	PREVIOUS_REVISION_ID
18885	24	926580	2010-03-23 15:10:33	uschindler	remove properties	0	1	18884

Figure 6: Removed trunk folder

The record was stop on the date 2010-03-23. However, the latest bug we retrieved from Jira was as shown in Figure 7:

ID	KEY	DESCRIPTION	BLOCKER	CRITICAL	MAJOR	MINOR	TRIVIAL	DATE_CREATED	DATE_RESOLVED	DATE_UPDATED	STATUS
1	LUCENE-3339	TestNRTThreads h	0	0	1	0	0	2011-07-25 01:45:08	2011-07-25 13:10:59	2011-07-25 13:10:59	Resolved

Figure 7: Bug information

The information shows that there was something between the date 2010-03-23 and 2011-07-25 so that there were no bugs updated from the date 2010-03-23 until now. Hence, we checked the revisions table again, and saw the very last record in this table as shown in Figure 8:

ID	SYSTEM_ID	REVISION_NUMBER	DATE	AUTHOR	COMMIT_MESSAGE	META_MODEL_COMPLETE	CHANGE_DATA_COMPLETE	PREVIOUS_REVISION_ID
18884	24	926577	2010-03-23 15:01:35	uschindler	Remove old trunk folder	0	1	18883
18885	24	926580	2010-03-23 15:10:33	uschindler	remove properties	0	1	18884

Figure 8: Bug information

The reason was identified is that, at the date 2010-03-23, they removed the old trunk folder and moved into the new trunk folder. Therefore, when we retrieved revisions information from the SVN repository, the last record of revision was on 2010-03-23, while the last record of bug information was on 2011-07-25. As a result, a

[9] Apache Lucene Java http://projects.apache.org/projects/lucene_java.html (last visited on July 2011)

new round of running the tool EvAn was defined. We used that tool to extract revisions information again, but with the new link to new trunk folder of project Apache Lucene Java.

In the second round of running, we retrieved 1614 new revisions, from the last date of the old trunk until the date we retrieved the bugs' information. Finally, we retrieved all updated revisions of Lucene Java project up to the date the latest bug found. The matching strategy was re-defined.

3.2.3 Revisions matching strategy

At the first step, we defined a matching strategy by creating a SQL query that used string comparison operator "LIKE" between the defect's KEY (e.g. LUCENE-3339) and the commit message column in revision entities table in order to find whether defect's KEY is a part of commit message, as follow:

SELECT def.KEY, def.DESCRIPTION, rev.ID, rev.COMMIT_MESSAGE

FROM `defect_entities` as def, `revision_entities` as rev

WHERE rev.COMMIT_MESSAGE LIKE concat('%', def.KEY, '%');

The meaning of this query command is to find a revision with a commit message that contains defect's KEY. But, the result was not as good as the expectations. There were so many duplicate records of revisions with the same defect's KEY but with different and incorrect relevant commit message. For instance, with the defect's KEY LUCENE-14, this query returned the list of revisions with the commit messages that contained all the KEYs started with LUCENE-14, e.g. LUCENE-14, LUCENCE-143, LUCENE-1416, etc. An example is shown in Figure 9:

Defect's KEY	Defect's description	Revision's ID	Commit message
LUCENE-14	QueryParser not handling Toker	17447	LUCENE-1404: fixed NPE in NearSpans
LUCENE-14	QueryParser not handling Toker	17448	LUCENE-1400: add rat-sources target t
LUCENE-14	QueryParser not handling Toker	17449	LUCENE-1402: make CheckIndex back
LUCENE-14	QueryParser not handling Toker	17450	LUCENE-1401: remove new deprecate
LUCENE-14	QueryParser not handling Toker	17452	LUCENE-1401: fix accidentally lost 'cre
LUCENE-14	QueryParser not handling Toker	17467	LUCENE-1412: clarify Directory.copy ja
LUCENE-14	QueryParser not handling Toker	17468	LUCENE-1414: increase max heap size
LUCENE-14	QueryParser not handling Toker	17469	LUCENE-1415: MultiPhraseQuery has i
LUCENE-14	QueryParser not handling Toker	17474	LUCENE-1419: Add expert API to set cu
LUCENE-14	QueryParser not handling Toker	17476	LUCENE-1423
LUCENE-14	QueryParser not handling Toker	17477	LUCENE-1416: don't fail contrib/ant's
LUCENE-14	QueryParser not handling Toker	17478	LUCENE-1411: add expert API to Index
LUCENE-14	QueryParser not handling Toker	17480	LUCENE-1406. Added Arabic stemmir
LUCENE-14	QueryParser not handling Toker	17483	LUCENE-1426: next steps towards flexi

Figure 9: Revisions information

Hence, this query command was not good and weak. A new SQL query was defined. We used regular expression metacharacters [[:<:]] and [[:>:]]. The metacharacter [[:<:]] is supposed to match the beginning of the words and the other is used to match ending of the words. The query command was refined as follow:

SELECT def.KEY, def.DESCRIPTION, rev.ID, rev.COMMIT_MESSAGE

FROM `defect_entities` as def, `revision_entities` as rev

WHERE rev.COMMIT_MESSAGE LIKE concat('[[:<:]]', def.KEY, '[[:>:]]');

This query command returned almost every single revision in the revision entities table regardless to the defect's KEY. For instance, with one defect's KEY, this query matched almost all the records in the table revision entities. This was a confusion and also not what we expected. A proper and more concrete result was needed instead of so a third attempt of refining the SQL query was carried out. New regular expression metacharacters was used.

The main intension here was that we intended to match the revision with commit message that exactly matches the defect's KEY. As an example, with the defect's KEY LUCENE-14, we only wanted to find a revision with commit message that contains exactly this phrase "LUCENE-14", not LUCENE-149, etc. This meant, we needed to exclude irrelevant numbers out of LUCENE-14.

There is another metacharacter that is appropriate to this, i.e. [^0-9]. This metacharacter means the query will return everything but excluded number from 0 to 9. For a more concrete explanation, the refined query command is as follow:

SELECT def.KEY, def.DESCRIPTION, rev.ID, rev.COMMIT_MESSAGE

FROM `defect_entities` as def, `revision_entities` as rev

WHERE rev.COMMIT_MESSAGE regexp concat(def.KEY, '[^0-9]')

ORDER BY def.KEY ASC;

The purpose of this SQL query was to intend to get the revisions with a commit message that contains exactly defect's KEY. As the result, this refined SQL query returned 861 results. The list of revisions that matched with the defect's KEYs is shown in Figure 10:

Defect's KEY	Defect's description	Revision's ID	Commit message
LUCENE-1640	MockRAMDirectory (used only b	17770	LUCENE-1640: fix MockRAMDir's inter
LUCENE-1645	Deleted documents are visible a	17775	LUCENE-1645: clone child SegmentRe
LUCENE-1648	when you clone or reopen an In	17777	LUCENE-1648: carry over hasChanges/
LUCENE-1647	IndexReader.undeleteAll can me	17781	LUCENE-1647: fix case where IndexRea
LUCENE-1655	remove 1.5 only unit test code th	17782	LUCENE-1655: get unit tests back to 1.
LUCENE-1666	Constants causing NullPointerE	17797	LUCENE-1666: if JAR has null impl vers
LUCENE-1666	Constants causing NullPointerE	17798	LUCENE-1666: use LucenePackage to
LUCENE-1681	DocValues infinite loop caused b	17866	LUCENE-1681: Fix infinite loop caused
LUCENE-1639	intermittent failure in TestIndex\	17872	LUCENE-1639: fix case where doc store
LUCENE-1700	LogMergePolicy.findMergesToE	17873	LUCENE-1700: make sure expungeDele
LUCENE-1715	DirectoryIndexReader finalize() h	17883	LUCENE-1715: remove finalize from In
LUCENE-1714	WriteLineDocTask incorrectly nc	17884	LUCENE-1714: fix WriteLineDocTask tc
LUCENE-1715	DirectoryIndexReader finalize() h	17899	LUCENE-1715: null out a few things or
LUCENE-1734	CharReader should delegate rese	17930	LUCENE-1734: CharReader should dele
LUCENE-1730	TrecContentSource should use a	17932	LUCENE-1730: Fix TrecContentSource
LUCENE-1717	IndexWriter does not properly ac	17938	LUCENE-1717: properly account for RA

Figure 10: Revision information

But, there are still 99 bugs that could not be matched. We checked the revision entities table again and found out the same reason as we described above. Somehow, when developers fixed bugs and committed on SVN repository, they did not include the defect's KEY into the commit message; so we were unable to know to which bug the revision belonged. With the matched revision of 861 out of 960 bugs, we assumed that this number was acceptable and we could ignore the 99 unmatched bugs.

In order to prepare for the evaluation afterwards, we created a new table in database that stored the information of bug's ID, reference revision's ID where bug was fixed, and previous revision's ID that might have contained bug already and should be checked and compared with bug-fixed revision.

When the revisions' information was ready, the next step was to collect classes' information, including both bug-fixed classes and bug-free classes. As a result, there were 1386 classes that had bugs found and fixed. In other words, 1386 classes were touched to fix 861 bugs. The last step was to collect the same number of bug-free classes. Bug-free classes are all the classes, which are not related to any fixed-bug classes on SVN. In other words, they are not touched to fix any bug. The number of bug-free classes was chosen randomly from the bug-free classes in database. This assumption is necessary to find the difference between bug-fixed and bug-free classes, therefore; it would be the base of knowledge to propose the models.

3.2.4 Conclusion on the data collection

As a conclusion, the training data had **2772** sets in the samples, including 1386 bug-fixed classes, which fixed 861 bugs, and 1386 bug-free classes. Bug-fixed classes are classes that involved or touched in order to fix all the bugs found. Bug-free classes are classes that contain no bug. They were picked randomly from the number of classes that contained no bug from the database. Afterwards, this training data was used to build a model that will be described in the next chapter.

4 EXPERIMENTAL MODEL AND DEFINITION AND RESULTS

In this chapter, the prediction models and their evaluations are described. Regression analysis and logistic analysis were carried out to find the promising models for predicting the defect density of one class and predicting whether one class has bug or no bug. This chapter is the result of the fifth phase (i.e., Correlation between indicators and defect density) in the methodology of this thesis.

4.1 Linear Regression Analysis Annotation

Linear regression analysis was conducted based on the data set we have retrieved in phase 4. The regression analysis was performed since we wanted to build a model to predict the value of a variable based on the value of other variables. In the scope of this evaluation, the different metric values retrieved (see the section 2.2.3) were used to predict the defect number of one class. Variables that were used to predict the other variable's value is called independent variable or predictor variable, sometimes. In this case, independent variables were the different metric values. On the other hand, the dependent variable or outcome variable was the one that would be predicted. In this scope, dependent variable is defect density of one class.

For regression analysis, linear regression analysis was chosen. It is an approach to estimate the coefficients of a linear equation. It can involve one or more independent variables and is comparably the best to predict the value of dependent value. Linear regression is suited in such case when given a variable Y and a set of variable X_1, X_2, …, X_n that could be related to value of Y; linear regression can be applied to clarify the relationship between Y and X_i; it's also used to verify which X_i may have strong correlation to Y, and which may be not at all.

4.1.1 Data preparation

The data for this evaluation were collected in the Lucene project, which divided into two categories: bug-fixed classes and bug-free classes. Afterwards, several metrics were calculated for those classes. The list of metrics is described below (see Appendix A):

- Class level: NOPA, AMW, NOM, ATFDClass, LOCClass, NOAM, WOC, WMC, TCC.
- Method level: MAXNESTING, NOAV, CC, CYCLO, CM.

For those metrics at the method level, one class can have more than one method. As a result, these metrics values were converted to class level by calculating their average values and sum values of all methods in one class. Then, these metrics values were considered as metrics values at class level.

Independent variables and dependent variable, in this case, are the different metric values and defect density of one class respectively, and these are quantitative. Dependent variable is simple that a variable depends on independent variable(s). For example, defect density that one class may contain is dependent on metrics lines of code of that class (LOCClass) or number of methods of that class (NOM), etc. These independent variables and dependent variable are nominal variables since their values do not have intrinsic order.

Table 6 shows the average values of these metrics between bug-fixed classes and bug-free classes.

	Bug-fixed classes	Bug-free classess
NOPA	0.0029	0.0160
AMW	1.9877	1.9941
NOM	19.5253	9.3563
ATFDClass	5.8023	2.1473
LOCClass	308.3355	111.2563
NOAM	1.4221	0.3775
WOC	-0.6847	-0.7870
WMC	47.1494	20.6387
TCC	0.0418	0.0498
MAXNESTING_Avg	0.6239	0.4601
MAXNESTING_Sum	15.3059	5.4333
NOAV_Avg	4.2671	3.3437
NOAV_Sum	88.6227	34.4445
CC_Avg	0.8298	0.5591
CC_Sum	28.1212	8.1757
CYCLO_Avg	2.1149	2.0986
CYCLO_Sum	48.8160	21.2998
CM_Avg	1.5123	0.8732
CM_Sum	60.4545	15.0798

Table 6: Average metrics' values between bug-fixed classes and bug-free classes

As conclusion can be drawn beforehand, this table tells some important information about the potential of which metrics could be contributed to the prediction model. For instance, the metric TCC has no big difference between the values of bug-fixed classes and bug-free classes, e.g., 0.0418 and 0.0498 respectively. Whereas, the metric LOCCLASS has a big different between the values of bug-fixed classes and bug-free classes, e.g., 308.3355 and 111.2563 respectively.

4.1.2 Assumption

For regression analysis, we made some assumptions that are described below:

- All variables are measured at interval or ratio level, i.e., independent variables and dependent variable are numerical value.
- All variables are approximately normally distributed, i.e., the variance of distribution of dependent variable should be constant for all values of independent variables.
- There is a linear relationship between variables.

4.1.3 Procedure

IBM SPSS tool[10] was used to support for the linear regression analysis. The analysis was conducted to the data set of all classes that contained bugs. The inputs

[10] IBM SPSS tool http://www-01.ibm.com/software/analytics/spss/ (last visited on November 2011)

were defect density of classes as dependent variable and 19 different metrics values of these classes as independent variables.

The defect_density was added to the Dependent section as dependent variable in this analysis. In the Independent(s) list, all of the metrics were added. Notice that, in this regression, in the Method list, "Enter" was chosen. It means that all the independent variables were added to the regression model, regardless on they were significant or not.

SPSS tool generated a few tables as the result of linear regression, which were output of the analysis.

4.1.4 Annotated Linear Regression Analysis

As presented above, SPSS generated some tables for linear regression. In this section, we introduce the important tables. They are the Model Summary table and Coefficients table which are described below.

Model Summary table: This table provides R, R^2 value, adjusted R^2 value and the standard error. This table helps to find how well data is fitted by the model. R is the value of observed and predicted values of the dependent variable. R's values are from -1 to 1, which indicates positive or negative relationship. The absolute value of R shows the strength of the relationship. The larger the absolute value is, the stronger the relationship.

R^2 is the percentage of variation in the dependent variable which is described by the regression model. R^2 values are from 0 to 1. The smaller R^2 value is, the less the model fits the data.

Adjusted R^2 value tried to fix R^2 value so that it is reflected more closely to the fit of the model in the population. By using R^2 value, it helps to determine which model is best fit.

Coefficients table: This table provides information related to each predictor variable. It can be used to get the information necessary to predict dependent variable from independent variables.

ANOVA table: This table indicates whether the regression model predicts the outcome or the dependent variable significantly well or not.

The following section describes the result of multiple regression analysis of dependent variable on independent variables.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.465[a]	.216	.211	.912

a. Predictors: (Constant), CM_Sum, NOPA, NOAV_Avg, WOC, TCC, CC_Avg, CYCLO_Avg, LOCClass, MAXNESTING_Avg, NOAM, ATFDClass, NOM, WMC, CM_Avg, AMW, MAXNESTING_Sum, NOAV_Sum, CC_Sum, CYCLO_Sum

b. Dependent Variable: defect_density

Table 7: Model Summary of Linear Regression Output

The first table is Model Summary table, shown in Table 7. It describes the overall model fit. The R column is the square root of R^2 and it is the correlation between the observed and predicted values of dependent variable (defect density of one class). In this case, R equals 0.465, this value indicates a good correlation between observed and predicted values.

The R^2 indicates the amount of variance in the dependent variable (defect density) that can be predicted from the independent variable (NOPA, AMW, NOM, etc.). R^2 equals 0.216, which describes that 21.6% of the variance in defect density that could be predicted by the independent variable. However, it has to be taken into account that this value is an overall measure of the strength of the association; therefore, it does not show the extent of any particular independent variable, which is associated with the dependent variable.

ANOVA[a]

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	631.324	19	33.228	39.911	.000[b]
	Residual	2291.168	2752	.833		
	Total	2922.492	2771			

a. Dependent Variable: defect_density

b. Predictors: (Constant), CM_Sum, NOPA, NOAV_Avg, WOC, TCC, CC_Avg, CYCLO_Avg, LOCClass, MAXNESTING_Avg, NOAM, ATFDClass, NOM, WMC, CM_Avg, AMW, MAXNESTING_Sum, NOAV_Sum, CC_Sum, CYCLO_Sum

Table 8: ANOVA of Linear Regression Output

The next table is ANOVA table, shown in Table 8. Regression, Residual and Total are the sources of variance in the dependent variable. The Total variance consists of the variance that can be explained by the independent variables (Regression) and the one, which is not described by the independent variables (Residual). In addition, Sum of Squares for Total is a result of the sum of Sum of Squares for Regression and Residual. This value indicates the fact that Total is divided into Regression and Residual variance.

Moreover, R^2 in the Model Summary table can be computed by the following formula:

$$R^2 = \frac{Sum\ of\ Squares\ for\ Regression}{Sum\ of\ Squares\ for\ Total} \quad (1)$$

$$R^2 = \frac{631.324}{2922.492} = 0.216 \quad (2)$$

The reason is that R^2 is the proportion of the variance, which explained is by the independent variables; therefore, R^2 can be yielded by applying that formula.

The df column is the degree of freedom that associated with the source of variance. The total df can be computed by N – 1, where N is the number of cases. In this case, there were 2772 cases, so the total DF is 2771. The df value for the model (Regression) can be computed by k -1, where k is number of independent variables. In this case, there were 19 independent variables in the model, but the intercept was

automatically included in the model where the df value for the model is 19. In addition, the df value for Residual is equal the df value for Total minus the DF model; hence, the df value for Residual is 2752.

The Mean Square column is Mean Squares values, which are computed by the Sum of Squares divided by their respective df value. These computations support the computation of the F ratio or F-statistic in the F column, which is the Mean Square for Regression divided by the Mean Square for Residual.

The **p-value** (located in the Sig. column) is associated with F is very small (0.000). These values are used to answer the question, "Can the independent variables reliably predict the dependent variable?". Afterwards, the p-value is compared to the significance level of:

α = 0.05

In case that the p-value is less than α, the answer would be: "The independent variables can reliably predict the dependent variable". In this case, p-value is equal 0.000, which is smaller than α; therefore, the conclusion can be made up that the group of independent variables NOPA, AMW, NOM, etc. can be used to predict the number of defects in one class (dependent variable).

The last table is Coefficients table, show in Table 9. This table represents the parameter estimates. The Model column shows the predictor variables such as NOPA, AMW, NOM, etc. The first value Constant represents the constant or is called the Y intercept, which is the height of regression line when it crosses the Y axis. In other words, this constant value is the value of the number of defects of one class when all other variables are zero. But this case seldom happens since zero is not a value that variables may carry.

The B column is the values of regression equation for predicting the dependent variable from independent variables. They were measured in their natural units; hence, they are called Unstandardized Coefficients. However, they cannot be compared with one another to examine which one has a strong influence in the model because they were measured in different scales. For instance, it would be incorrect to compare the value of NOM and the value of LOCClass. The regression equation can be formed as follow:

$$Y = b_0 + b_1 * x_1 + b_2 * x_2 + b_3 * x_3 + \cdots + b_n * x_n$$

The column of parameter estimates, i.e. coefficient values, offers the values for b0, b1, b2 … bn for this equation. In terms of the variables used in this evaluation, the equation is derived as below:

$$\begin{aligned} defect\ density \\ &= 0.241 - (0.088 * \mathrm{NOPA}) + 0.045 * \mathrm{AMW} + 0.020 * \mathrm{NOM} + 0.040 \\ &* \mathrm{ATFDC} + 0.001 * \mathrm{LOCC} + 0.053 * \mathrm{NOAM} + 0.017 * \mathrm{WOC} - 0.003 \\ &* \mathrm{WMC} - 0.089 * \mathrm{TCC} + 0.011 * \mathrm{MAXNESTING_{Avg}} - 0.004 \\ &* \mathrm{MAXNESTING_{Sum}} + 0.041 * \mathrm{NOAV_{Avg}} - 0.003 * \mathrm{NOAV_{Sum}} - 0.267 \\ &* \mathrm{CC_{Avg}} + 0.001 * \mathrm{CC_{Sum}} - 0.039 * \mathrm{CYCLO_{Avg}} - (2.682 * 10^{-5}) \\ &* \mathrm{CYCLO_{Sum}} + 0.205 * \mathrm{CM_{Avg}} - 0.002 * \mathrm{CM_{Sum}} \end{aligned}$$

(3)

Where:

- ATFDC is ATFD value of class
- LOCC is LOC value of class

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	.241	.050		4.768	.000
	NOPA	-.088	.021	-.082	-4.298	.000
	AMW	.045	.037	.107	1.220	.223
	NOM	.020	.004	.406	5.699	.000
	ATFDClass	.040	.005	.294	8.589	.000
	LOCClass	.001	.000	.333	3.937	.000
	NOAM	.053	.010	.181	5.575	.000
	WOC	.017	.025	.013	.677	.498
	WMC	-.003	.003	-.208	-.866	.387
	TCC	-.089	.039	-.049	-2.308	.021
	MAXNESTING_Avg	.011	.059	.007	.184	.854
	MAXNESTING_Sum	-.004	.005	-.088	-.821	.412
	NOAV_Avg	.041	.012	.110	3.342	.001
	NOAV_Sum	-.003	.001	-.341	-2.283	.023
	CC_Avg	-.267	.061	-.315	-4.349	.000
	CC_Sum	.001	.003	.065	.336	.737
	CYCLO_Avg	-.039	.042	-.086	-.910	.363
	CYCLO_Sum	-2.682E-005	.003	-.002	-.008	.994
	CM_Avg	.205	.035	.432	5.814	.000
	CM_Sum	-.002	.001	-.345	-1.936	.053

a. Dependent Variable: defect_density

Table 9: Coefficients of Linear Regression Output

These estimates indicate the relationship between the dependent variable and the independent variables. They explain the amount of increase or decrease in the defect_density score that would be predicted by a one-unit increase (positive value of coefficient) or decrease (negative value of coefficient) in the predictor. For example:

- NOPA: The coefficient is -0.088; therefore, for every unit decreases in NOPA score, a 0.088 unit decreases in defect_density is predicted, holding all other variables constant. Because it is a linear model, it is not necessary to indicate what value holds the other variables constant.
- AMW: The coefficient is 0.045; hence, for every unit increases in AMW score, there is a 0.045 unit increases in defect_density, holding all the other variables constant.
- The annotations for the other metrics are the same, based on the positive or negative coefficient values.

The Std. Error column is the standard errors associated with coefficients. These values are used to examine whether the parameter is significantly different from zero by dividing the parameter by the standard error, yielding t-value (located in t column). The t-value and p-value (located in the Sig. column) are used to explain whether the predictor is significant statistic or not. As such, the conclusions about coefficients are described below:

- The coefficients for NOPA, NOM, AFTDClass, LOCClass, NOAM, TCC, NOAV_Avg, NOAV_Sum, CC_Avg, CM_Avg, CM_Sum are statistically significantly different from zero because their p-value are smaller than significant level α.
- The coefficients for AMW, WOC, WMC, MAXNESTING_Avg, MAXNESTING_Sum, CC_Sum, CYCLO_Avg, CYCLO_Sum are not statistically significant different from 0 because their p-value are greater than significant level α.

However, in the model (3), it consists of many insignificant predictors; consequently, that would make this model not accurate and not applicable. Therefore, another regression with the method Stepwise was carried out to eliminate these insignificant predictors from the model.

4.2 Stepwise Linear Regression Analysis Annotation

In this section, regression equation using Stepwise method is described. Stepwise Linear Regression is a method using for multiple regression variables that simultaneously removes all variables that are not important and significant [48]. The workflow of Stepwise Linear Regression is shown in the Figure 11.

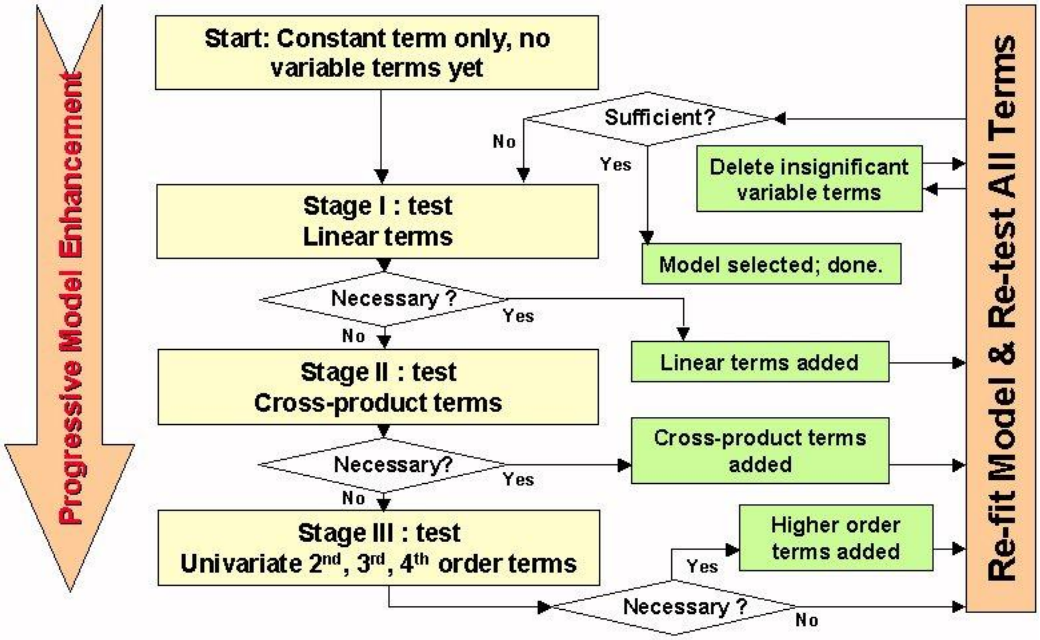

Figure 11: Stepwise Linear Regression [49]

The main approaches of Stepwise regression are [50]:

- **Forward selection**: this step involves starting without any variables in the model, trying out every single variable and including them in the model if they are statistically significant.

- **Backward elimination**: this step involves starting with all the potential variables and testing them one by one for statistical significance, and eliminating any variable that is not significant.
- Methods that are a combination of the above approaches, which examine at each stage for variables to be included or excluded.

4.2.1 Data preparation

The independent variables and dependent variable use in the Stepwise regression analysis were the same set as the linear regression. All the variables were used to find the most significant ones.

4.2.2 Procedure

IBM SPSS tool was used to carry out the Stepwise regression.

The dependent variable was defect density of one class. The independent variables were all the metrics. The main thing was that in "Method" list, Stepwise was chosen. Stepwise regression essentially does multiple regressions a number of times, at each time removing the weakest correlated variable. At the end, the model was left with the variables that could explain the distribution best. The annotation of the results is explained in next section.

4.2.3 Annotated Stepwise Regression Analysis

The first table is Variables Entered/Removed[11]. This table indicates the variables that were used to build the models. It shows the order in which the variables were entered and removed from the model. In this case, 18 variables were entered and 3 variables were removed (*WOC, MAXNESTING_Sum, CC_Sum*).

[11] See Appendix B.

Variables Entered/Removed[a]

Model	Variables Entered	Variables Removed	Method
1	ATFDClass	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
2	NOM	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
3	WMC	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
4	NOAV_Avg	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).

Figure 12: Variables Entered/Removed table

The next table is the Model Summary, shown in Table 10. This table gives the details of the overall correlation between variables left in the models and the dependent variable. As introduced above, the last model would contain variables that could explain the distribution best. Hence, in the model 18, there was 21.2% variation in the dependent variable that could be explained using the independent variables included in model 18.

Model Summary[s]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.405[a]	.164	.164	.939
2	.416[b]	.173	.173	.934
3	.424[c]	.180	.179	.931
4	.431[d]	.186	.184	.927
5	.434[e]	.189	.187	.926
6	.437[f]	.191	.189	.925
7	.439[g]	.193	.191	.924

8	.443[h]	.196	.194	.922
9	.448[i]	.201	.198	.920
10	.447[j]	.200	.198	.920
11	.455[k]	.207	.205	.916
12	.457[l]	.209	.206	.915
13	.458[m]	.210	.207	.915
14	.460[n]	.212	.209	.914
15	.463[o]	.214	.210	.913
16	.462[p]	.214	.210	.913
17	.464[q]	.215	.211	.912
18	.464[r]	.215	.212	.912

a. Predictors: (Constant), ATFDClass

b. Predictors: (Constant), ATFDClass, NOM

c. Predictors: (Constant), ATFDClass, NOM, WMC

d. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg

e. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA

f. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, WOC

g. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, WOC, CC_Sum

h. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, WOC, CC_Sum, NOAM

i. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, WOC, CC_Sum, NOAM, CM_Avg

j. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg

k. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg

l. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg, TCC

m. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg, TCC, MAXNESTING_Sum

n. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg, TCC, MAXNESTING_Sum, LOCClass

o. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg, TCC, MAXNESTING_Sum, LOCClass, NOAV_Sum

p. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg, TCC, LOCClass, NOAV_Sum

q. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, CC_Sum, NOAM, CM_Avg, CC_Avg, TCC, LOCClass, NOAV_Sum, CM_Sum

r. Predictors: (Constant), ATFDClass, NOM, WMC, NOAV_Avg, NOPA, NOAM, CM_Avg, CC_Avg, TCC, LOCClass, NOAV_Sum, CM_Sum

s. Dependent Variable: defect_density

Table 10: The Model Summary

The important table is the Coefficients[12] from which the Stepwise regression equation was formed. This table shows the linear regression equation coefficients of the various model variables. These values were explained in the linear regression annotation. The B values are the coefficients for each variable, i.e. they are values that the variable's data should be multiplied in the final linear equation in order for the defect density to be predicted.

In the model 18, all the variables left are significant (their p-values are smaller than the significant level α) except TCC metric; therefore, this metric was excluded from the model. Consequently, the regression model was formed as follow:

$$defect\ density = 0.222 + 0.041 * ATFDC + 0.021 * NOM - 0.003 * WMC + 0.048 * NOAV_{Avg} - 0.081 * NOPA + 0.057 * NOAM + 0.209 * CM_{Avg} - 0.273 * CC_{Avg} + 0.001 * LOCC - 0.004 * NOAV_{Sum} - 0.002 * CM_{Sum} \quad (4)$$

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
18	(Constant)	.222	.041		5.424	.000
	ATFDClass	.041	.005	.300	8.925	.000
	NOM	.021	.003	.415	6.311	.000
	WMC	-.003	.000	-.192	-5.240	.000
	NOAV_Avg	.048	.008	.127	5.621	.000
	NOPA	-.081	.020	-.076	-4.144	.000
	NOAM	.057	.009	.193	6.596	.000
	CM_Avg	.209	.032	.441	6.464	.000
	CC_Avg	-.273	.050	-.321	-5.492	.000
	TCC	-.062	.034	-.034	-1.806	.071
	LOCClass	.001	.000	.338	4.262	.000
	NOAV_Sum	-.004	.001	-.461	-4.377	.000
	CM_Sum	-.002	.000	-.295	-7.525	.000

a. Dependent Variable: defect_density

Figure 13: Coefficients Table

[12] See Appendix B

4.3 Logistic Regression Analysis Annotation

In this section, prediction model and its evaluation are described by using Logistic Regression. Logistic regression was carried out to support the regression analysis to predict whether one class may contain bug or not. Logistic regression, according to SPSS Help, is useful for the situations in which researchers want to be able to predict the presence or absence of a characteristic or outcome based on the values of a set of predictor variables. Basically, logistic regression is similar to a linear regression model, but it is more suited to models where the dependent variable is dichotomous. In other words, logistic regression regresses a dichotomous dependent variable (target) on a set of independent variables (predictor).

Logistic regression was applied to find a model in order to predict whether a class may contain a bug. This logistic regression model can support the linear regression model to predict the number of bugs one class may have. This information is important so that one organization may know which class has defect prone and focus on it. Therefore, it can help them save time spending on quality assurance activities.

4.3.1 Data preparation

In logistic regression, the same sample population in linear regression was used. In other words, independent variables or covariates (term used in logistic regression) were values of several metrics. Notice that, independent variables used in logistic regression were all the independent variables left in the equation (4).

Because the data sample did not have a suitable dichotomous variable to use as dependent variable, one was created based on the bug status of classes. For classes that contained bugs, a flag '1' was set to them. For the rest of bug-free classes, a flag '0' was assigned to them. This step made dependent variable dichotomous. The dichotomous dependent variable was named 'defect prone'.

4.3.2 Procedure

SPSS tool was used to performed logistic regression. The dependent variable was defect prone of one class. The covariates were values of different metrics.

4.3.3 Annotated Logistic Regression Output

SPSS generated some tables for the output of logistic regression analysis. In this part, the annotations of these tables are described. The tables would tell about the cases that were included or excluded from the analysis, coding of the dichotomous dependent variable, relationship between dependent variable and independent variables and prediction model.

The first table is Case Processing Summary, shown in Table 11. This table introduces the cases that are in the analysis. N column and Percent show the number of cases in each category, including analysis, missing cases and the total. In this table, there were 2772 cases included in the analysis. Because the analysis did not have missing data, this row also indicated the total number of cases, consisting of 1386 cases, which are classes containing bugs and 1386 cases which are classes that were bug free.

The row Missing Cases indicates the cases that were excluded from the analysis. By default, the SPSS tool will do a listwise deletion of missing data. In other words, if there is any missing value for any variable in the model, the entire case will be

excluded from the analysis. In this model, all the value were there; therefore, there were no missing cases.

Case Processing Summary

Unweighted Cases[a]		N	Percent
Selected Cases	Included in Analysis	2772	100.0
	Missing Cases	0	.0
	Total	2772	100.0
Unselected Cases		0	.0
Total		2772	100.0

a. If weight is in effect, see classification table for the total number of cases.

Table 11: Case Processing Summary of Logistic Regression Output

The next table is the Dependent Variable Encoding, shown in Table 12. This table indicates the encoding of dichotomous values of dependent variable. As described in the Data Preparation section, dichotomous dependent variable had two values 0 and 1, which indicated bug free and bug fixed classes respectively. To be used in logistic regression by SPSS, e.g., Internal Value column of this table, these values were encoded and by a chance, they were the same with original value.

Dependent Variable Encoding

Original Value	Internal Value
0	0
1	1

Table 12: Dependent Variable Encoding of Logistic Regression Output

4.3.3.1 Null hypothesis

The next part of the output describes a "null model" or null hypothesis, which is a model in which all predictors were excluded and just the intercept was included.

H_0: There is no relationship between predictors and dependent variable

The first table is Classification Table, shown in Table 13. Observed column indicates the number of 0's (bug free classes) and 1's (bug fixed classes) that are observed in the dependent variable. Predicted column, in this null model, predicts that all the cases were 1 (bug fixed classes) on the dependent variable. The Overall Percentage row describes the percentage of cases for which the dependent variable was correctly predicted by given the null model:

50.0 = 1386/2772

Classification Table[a,b]

		Observed	Predicted		
			defect_prone		Percentage Correct
			0	1	
Step 0	defect_prone	0	0	1386	.0
		1	0	1386	100.0
	Overall Percentage				50.0

a. Constant is included in the model.

b. The cut value is .500

Table 13: Classification Table of Logistic Regression Output

In the table Variables in the Equation, shown in Table 14, the B column is the coefficient for the constant, which is also called intercept in the null model. The Sig. column is the p-value. In statistical significance testing, p-value is the probability of obtaining a test statistic at least as extreme as the one that was actually observed. With a p-value is 1, it would be incorrect conclude that there is no effect of metrics on defect prone. Therefore, the conclusion was that the constant cannot be 0.

Variables in the Equation

		B	S.E.	Wald	df	Sig.	Exp(B)
Step 0	Constant	.000	.038	.000	1	1.000	1.000

Table 14: Variables in the Equation

The last table in null model output is Variables not in the Equation, shown in Table 15. Because this is a null model, all the predictors are in it. The Score column is a score that was used to predict where an independent variable might be significant in the model or not. Checking the p-values which are located in Sig. column, it's clear that each of the predictors would be statistically significant (where p-value is smaller than significance level $\alpha = 0.05$) except TCC.

Variables not in the Equation

			Score	df	Sig.
Step 0	Variables	ATFDClass	245.058	1	.000
		NOM	215.346	1	.000
		WMC	84.821	1	.000
		NOAV_Avg	95.851	1	.000
		NOPA	20.923	1	.000
		NOAM	69.844	1	.000
		CM_Avg	76.822	1	.000
		CC_Avg	33.521	1	.000
		LOCClass	181.484	1	.000
		NOAV_Sum	206.246	1	.000
		CM_Sum	75.556	1	.000
	Overall Statistics		477.659	11	.000

Table 15: Variables not in the Equation

4.3.3.2 Alternative hypothesis

The last part of the output is important because it defines the prediction model or alternative hypothesis.

H_1: There is a relationship between predictors and defect prone.

The first table is the Omnibus Tests of Model Coefficients, shown in Table 16, which is the overall test of the model. This is a model with all predictors in it. The Chi-square and Sig. column are the chi-quare statistics and its significance. In this case, all the p-value (located in Sig. column) are smaller than critical value of the significance level α, i.e. the model is statistically significant.

Omnibus Tests of Model Coefficients

		Chi-square	df	Sig.
Step 1	Step	714.492	11	.000
	Block	714.492	11	.000
	Model	714.492	11	.000

Table 16: Omnibus Tests of Model Coefficients of Logistic Regression Output

The next table is Classification Table, shown in Table 17. Observed column indicates the number of 0's and 1's that are observed in the dependent variable. On the other hand, the Predicted column indicates the predicted value of the dependent variable based on the full logistic regression model. This table has two categories of prediction. They are described below:

- How many cases are correctly predicted: 1122 cases are observed to be 0 and are accurately predicted to be 0; 785 cases are observed to be 1 and are accurately predicted to be 1.
- How many cases are not correctly predicted: 264 cases are observed to be 0 but are predicted to be 1; 601 cases are observed to be 1 but are predicted to be 0.

The Percentage Correct column indicates all the cases that are accurately predicted by the model, which is the alternative model. As a result, the percentage has increased from 50% for the null model to 68.8% for the alternative model.

Classification Table[a]

	Observed		Predicted		
			defect_prone		Percentage
			0	1	Correct
Step 1	defect_prone	0	1122	264	81.0
		1	601	785	56.6
	Overall Percentage				68.8

a. The cut value is .500

Table 17: Classification Table of Logistic Regression Output

The last table of this part is the Variables in the Equation, shown in Table 18. Because this is a full model, all the predictors are included in it. All values in B column are the values that are used in the logistic regression equation for predicting

the dependent variable (defect prone) from the independent variables (all the used metrics). The prediction equation is:

$$\log\left(\frac{p}{1-p}\right) = b_0 + b_1 * x_1 + b_2 * x_2 + b_3 * x_3 + \cdots + b_n * x_n$$

Where: p is the probability of being in the composition

Applied in the context of variables used in this evaluation, the logistic regression equation is derived below:

$$defect\ prone = -1.090 + 0.174 * \text{ATFDC} + 0.045 * \text{NOM} - 0.018 * \text{WMC} + 0.025 * \text{NOAV}_{\text{Avg}} - 0.441 * \text{NOPA} + 0.128 * \text{NOAM} + 0.665 * \text{CM}_{\text{Avg}} - 0.849 * \text{CC}_{\text{Avg}} + 0.007 * \text{LOCC} - 0.009 * \text{NOAV}_{\text{Sum}} - 0.006 * \text{CM}_{\text{Sum}}$$

(5)

This equation explains the relationship between the dependent variable and independent variables, where dependent variable is on the logit scale. It also describes the quantity of increase (if the sign of coefficient is positive) or decrease (if the sign of coefficient is negative) in the predicted log odds of defect prone = 1 that would be predicted by a one-unit increase (or decrease) in the predictor, which holding all the other predictors constant. Therefore, the equation would be annotated as below:

- NOPA – for every one-unit decrease (the sign of coefficient is negative) in NOPA metric score, it is supposed that a 0.441 decrease in the log-odds of defect_prone, holding all the others independent variables constant.
- NOM – for every one-unit increase (the sign of coefficient is positive) in NOM metric score, it is supposed that a 0.045 increase in the log-odds of defect_prone, holding all the others independent variables constant.
- The annotations are the same for the rest of other independent variables based on whether the sign of coefficient is positive or negative.
- The constant -1.090 is the expected value of the log-odds of defect_prone when all the predictors' values are equal to zero. However, this case rarely happens since zero is not a real value for a variable to take.

Variables in the Equation

		B	S.E.	Wald	df	Sig.	Exp(B)
Step 1[a]	ATFDClass	.174	.021	72.183	1	.000	1.190
	NOM	.045	.012	13.026	1	.000	1.046
	WMC	-.018	.002	62.328	1	.000	.982
	NOAV_Avg	.025	.026	.963	1	.326	1.025
	NOPA	-.441	.077	32.707	1	.000	.644
	NOAM	.128	.032	15.737	1	.000	1.137
	CM_Avg	.665	.121	30.213	1	.000	1.945
	CC_Avg	-.849	.161	27.790	1	.000	.428
	LOCClass	.007	.001	51.654	1	.000	1.007
	NOAV_Sum	-.009	.003	7.061	1	.008	.991
	CM_Sum	-.006	.002	5.530	1	.019	.994
	Constant	-1.090	.113	92.443	1	.000	.336

a. Variable(s) entered on step 1: ATFDClass, NOM, WMC, NOAV_Avg, NOPA, NOAM, CM_Avg, CC_Avg, LOCClass, NOAV_Sum, CM_Sum.

Table 18: Variables in the Equation of Logistic Regression Output

4.4 Conclusion on the experiment

As the result of the experiment, there are two proposed model. The first model is shown in the model equation (4):

$$defect\ density = 0.222 + 0.041 * \text{ATFDC} + 0.021 * \text{NOM} - 0.003 * \text{WMC} + 0.048 * \text{NOAV}_{\text{Avg}} - 0.081 * \text{NOPA} + 0.057 * \text{NOAM} + 0.209 * \text{CM}_{\text{Avg}} - 0.273 * \text{CC}_{\text{Avg}} + 0.001 * \text{LOCC} - 0.004 * \text{NOAV}_{\text{Sum}} - 0.002 * \text{CM}_{\text{Sum}} \quad (4)$$

It is used to predict the number of defects that a class may contain. It takes the input as the metrics' values of one class and returns the output as the number of defect that class may contain.

The second mode is shown in the model equation (5):

$$defect\ prone = -1.090 + 0.174 * \text{ATFDClass} + 0.045 * \text{NOM} - 0.018 * \text{WMC} + 0.025 * \text{NOAV}_{\text{Avg}} - 0.441 * \text{NOPA} + 0.128 * \text{NOAM} + 0.665 * \text{CM}_{\text{Avg}} - 0.849 * \text{CC}_{\text{Avg}} + 0.007 * \text{LOCClass} - 0.009 * \text{NOAV}_{\text{Sum}} - 0.006 * \text{CM}_{\text{Sum}} \quad (5)$$

It is used to predict if one class contains bug or no bug. This equation takes input as the metrics' values of one class and returns the output as the possibility that class may contain bug. If the output's value is positive, it indicates that class has no bug. If the output's value is negative, it shows that class has bug in it. For QA, this information is important to know whether one class has bug or not.

4.5 Variables

We selected the variables very carefully because this selection could affect the entire result of the experiment and the evaluation.

4.5.1 Independent variables

In this experiment, the independent variables are listed below:

- Selection of input parameters: project's selection for the experiment, tools, etc
- Context parameters: including product metrics or indicators were selected above.

4.5.2 Dependent variable

In our work, the dependent variable is the growth of defect between releases of the Lucene open source project or the defect's report of each release. This variable can affect to the evaluation.

5 MODEL VALIDATIONS

In this chapter, the validations of the models' equations (4) and (5) in Chapter 4 are described. The test data for validations, the scale used for the evaluation of the correctness of the equations and the comparison with the other model are introduced in details.

5.1 Test data for validations

The project Apache log4j[13] was used for these experimentation evaluations as the test data. Log4j is an experimental development branch for logging services designed for Java 5 and later. In order to prepare the test data for the validations, the collected bug information stored in EvAn's database was used.

An analysis based on these bug information was performed to look up the bug-fixed classes and bug-free classes for the validations. Consequently, 338 classes included bug-fixed and bug-free ones were selected. 169 classes contained bugs and were fixed for the Apache Log4j project. The other 169 bug-free classes were chosen randomly from the collection of bug-free classes (with the same assumption mentioned in Chapter 4). Hence, 338 sets in the samples were used as the test data for validation.

Each sample test data has the actual number of bugs, which that class contained. For the bug-fixed classes, the actual number of bugs was the bugs found and fixed for these classes. For the bug-free classes, the actual number of bugs was zero.

5.2 Execution of validations

After the test data was ready, all the metrics values required in the model's equation (4) were calculated. They are: ATFDClass, NOM, WMC, NOAV_Avg, NOPA, NOAM, CM_Avg, CC_Avg, TCC, LOCClass, NOAV_Sum, CM_Sum (see Appendix A).

Afterwards, the equation (4) was applied to the test data to calculate the predicted number of bugs for test samples (see the attached CD). A scale called Root Mean Square Error (RMSSE) was applied to evaluate the result of the model.

5.2.1 Root Mean Square Error (RMSSE)

RMSSE [51] is the scale frequently used to measure the differences between predicted values calculated by a model and actual values observed from the objects being modelled. RMSSE is the square root of mean square error as described in equation below:

$$\sqrt{\frac{(a_1 - c_1)^2 + (a_2 - c_2)^2 + \cdots + (a_n - c_n)^2}{n}} \quad (6)$$

Where:

- a_i is the actual values of the object being modelled
- c_i is the predicted values calculated by model

[13] Apache log4j http://logging.apache.org/log4j/

- n is the total number of object being observed

Since the literature considers RMSSE a good measure of accuracy, it is an ideal scale to evaluate the model. For this model, a_i is the actual number of defects of each class, and ci is the predicted number of defects of each class as calculated by model where n is equal 338 since there were 338 samples in the test data. Thus, the equation (6) was applied to the test data to calculate the differences between actual and predicted number of defects. Therefore, the value of RMSSE was:

RMSSE = 1.28

This value means there is 1.28 units difference between the actual values and the values predicted by using the model's equation (4). This value can also be considered as a promising value for the proposed model since it is almost of minimum difference. Figure 14 describes the distinction between the actual number of defects and the values predicted by applying the model equation (4). It shows the fitting between the actual and predicted number of defects.

According to the graph, there are about 160 classes that the model predicted the exact number of defect, i.e., the different value between actual and predicted value is zero. For the positive values in the X axis, they indicate that the actual number of defects is greater than the predicted number of defects. For instance, there are 60 classes that the actual number of defects is one unit greater than the defect number. For the negative values in the X axis, they indicate that the actual number of defects is less than the predicted number of defects. For example, there are 72 classes that the actual number of defects is one unit less than the predicted number of defects. The distinction can be explained that the missing defects has not found yet.

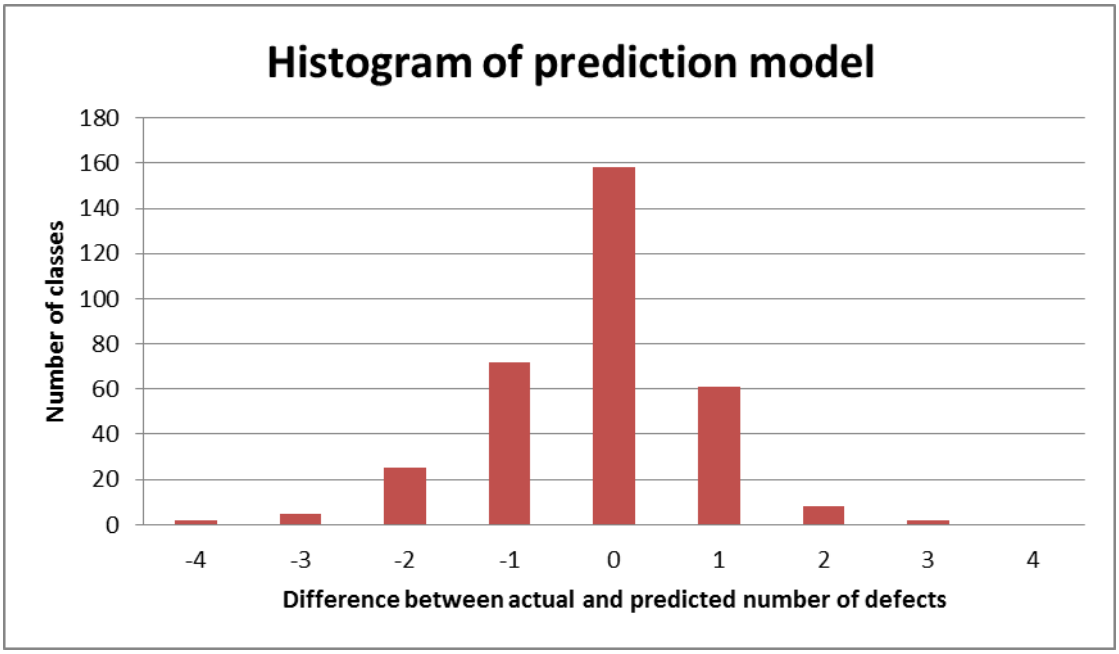

Figure 14: Applied model (4) to test data

However, in order to make it more convincible, it was compared with a well-known measure, named COQUALMO[14], from the COCOMO[15] family. COQUALMO is also called a quality model. COQUALMO takes input as LOC of one object

[14] COQUALMO http://sunset.usc.edu/csse/research/COQUALMO/ (last visited on Dec, 2011)
[15] COCOMO http://www.softstarsystems.com/overview.htm (last visited on Dec, 2011)

(software project, module, component, etc.) and returns output as the predicted number of defects of the object. In this measure, the objects were classes. Thus, COQUALMO measure was applied to compute the predicted number of defects for each class (see the attached CD).

Afterwards, the RMSSE value was also computed for the COQUALMO model in order to compare with the model's equation (4). Therefore, RMSSE value was:

RMSSE = 1.41

This value means there is 1.41 units difference between actual values and values predicted by using the COQUALMO model. As a comparison, RMSSE of model's equation (4) was 1.28; it indicated that model's equation (4) is as good as a well-known measure like COQUALMO.

5.2.2 Percentage of Correctness

In order to evaluate the accuracy of model's equation (5) – to predict whether one class contains bug or not, i.e. the output is binary value, the percentage of correctness scale was used. This scale returns the correctness of a model based on the number of correct answers out of total cases, as shown in the formula below:

$$\text{The percentage of correctness} = 100 * \frac{Number\ of\ correct\ answers}{Total\ number\ of\ cases}$$

Thus, the model's equation (5) was applied to the test data to compute defect-prone of each class, i.e. to predict whether one class contains bug or not (the result can be found in the attached CD). Afterwards, the number of correct answers was counted. As a result, there were 209 cases with correct answer out of 338 cases. Hence, the percentage of correctness could be computed as below:

$$\text{The percentage of correctness} = 100 * \frac{209}{338} = 62\%$$

As a comparison with the training data used to build the model, the correctness of training data was 68% and the correctness of test data was 62%. These values were acceptable since it is close to the value of training data.

5.3 Conclusion on the validations

In this chapter, each proposed model was compared to the trustworthy scale in order to assess the correctness of them.

To validate the correctness of the model equation (4), it was compared to the COQUALMO scale, which comes from the COCOMO family and is well-known as a good model to predict the number of defects based on LOC. Both the equation (4) and COQUALMO were applied to test data in order to calculate the number of defects for the test data.

Afterwards, both results were compared together by using the scale Root Mean Square Error (RMSSE) in order to assess the correctness the equation (4). As a result, the RMSSE values of the equation (4) and COQUALMO are 1.28 and 1.41 respectively. These numbers mean there are 1.28 units (1.41 units for COQUALMO) different between the actual number of defects and the predicted number of defect.

This result shows that the proposed model equation (4) is as good as the well-known model as COQUALMO for predicting the number of defect.

To validate the correctness of the proposed model equation (5), it was assessed by using the Percentage of Correctness scale. The equation (5) was applied to the test data to predict the defect-prone of classes. Afterwards, the Percentage of Correctness was applied by dividing the number of correct answers to the total number of test cases. As a result, the correctness of the equation is 62%. It is considered as a promising result as the correctness of training data is 68%.

6 THREATS TO VALIDITY

As in many empirical studies, there are some threats to validity in this thesis. In this section, we describe and address the appropriate threats to validity of data collection and analysis according to categories recommended by C Wohlin et.al [32].

6.1 Conclusion validity

The main threat with regards to conclude validity was the quality of collected data during the third phase of this study, which is the collected data for the experiment. As an attribute of the open source project, there is a possibility that some defect reported in current release are former defects reported in previous releases, which could not be perceived properly.

In addition, because of the open source project, there are many contributions from numerous people and different organizations; therefore, there is not a conventional way to fix bugs. For instance, when the bugs were fixed and committed to SVN, there is some missing information in the commit messages, e.g., bug's number or defect's key. As a result, we could not clarify to which defect this commit message belongs.

That was a threat we faced when we selected project Xerces2Java in the first step. In this project, there were so many commit messages without the defect's key in it; therefore, we could not conduct our experiment thoroughly. To overcome this threat, we decided to select another open source project, and we picked the Apache Lucene. But, as an attribute of an open source project, Lucene also has same threat as Xerces2Java, i.e. missing information in commit messages. However, the numbers of these commit messages were much less in the Lucene than in the Xerces2Java project. So, we decided to ignore them and take the rest of the qualified data for the experiment.

6.2 Internal validity

History

In this experiment, the metrics value such as cyclomatic complexity and lines of code were applied to the object of the experiment in order to propose a new model for defect prediction.

Maturation

One of the threats with regards to maturation is the publication biasness related to the numbers of published papers, articles and journals. This threat was ruled out in this thesis by selecting both positive and negative primary studies.

The object of the open source experiment should be trustworthy enough for the experiment. However, to overcome this threat, we selected the open source project Apache Lucene Java which was developed in 2004 and has garnered better results, thoughts and consideration from the users.

Instrumentation

In order to collect sufficient data for the experiment, we used Jira issues tracking system to retrieve defect information of the Apache Lucene project. Then, we used the tool EvAn to collect the defect information details to analyze.

6.3 Construct validity

In the literature review phase, we defined some search strings, which were made according to the suggestions of BTH librarians in the lecture note of the course Research Methodology [46].

In the experiment phase, we dealt with a difficulty situation, which was how to collect the number of defects of one release. Because the Lucene is an open source project, there is little summary information about the project. To overcome this threat, we scaled time between two releases and measured the number of defects for one release.

Thus, this ruled out any threat related to construct validity.

Restricted generalizability across construct

The prediction formula would be based on the issues report identified in the Lucene project. This solution might lead to some unintended side effects that, at the moment, are unforeseen and unknown.

In addition, we also considered the attributes of open source projects. Defect density of open source ones is easy to get ambiguities because there are a few number of defects density.

6.4 External validity

In this study, we proposed the prediction formula based only on the issues of tracking system and the issue reports we retrieved from the Lucene open source project. If we conduct an experiment with another software organization, the result would be different.

On the other hand, only one Lucene open source project is a negligible number; but it is a limited number of open source projects that is trustworthy enough to use for the experiment.

More importantly, the results came from the analysis of two open source Java systems. Thus, it is highly recommended not to generalize these results to other systems (e.g., closed source systems or systems that based on a different programming language) without analysing them to see if they are have some similarities to the ones being investigated in this thesis.

7 CONCLUSION AND FUTURE WORK

Quality assurance (QA) is an essential part in the software engineering development process. Within this process, QA activities are used to detect the performance of software project to validate whether it behaves as expected or not. QA activities are carried out during the software development life cycle. In addition, QA activities take part in reducing the risks of the software's quality. This point is counted on the software's budget. Therefore, accurate planning, launching and controlling QA activities are contributed to the success of software project.

Thus, QA activities consume time and cost of an organization's activities. One of the reasons is that QA activities may not focus on the potential defect-prone area. In fact, in some of the latest findings, researchers recommend that organizations should pay attention on QA activities in order to save time and cost. Additionally, researchers recommend that defect indicators are required to take part in this aspect.

Based on the above ideas, this thesis proposes new models by using the combination of software metrics as effective defect indicators. In order to build models, the open source project Lucene was used. Bug information of classes in the Lucene project were collected to use as training data to build models. The first model (4) is used to predict the number of defects in one class, which takes inputs as metrics' values of one class and returns output as values of predicted number of defects. The second model (5) is used to predict whether a class contains a bug or not.

Afterwards, the proposed models were validated and compared with the well-known measure COQUALMO to evaluate their accuracy and correctness. As a result, the models have a promising value when the model (4) has the results as good as the COQUALMO's result, with the accuracy of the model (5) being 62%.These results are promising to organizations so that the proposed models can be used to help improve the performance of QA activities in terms of time and cost.

In future work, the author would like to evaluate these models with some closed projects. Because the training data and test data used to build and validate the models were from open source projects. Due to some characteristics of the open source projects, i.e., the collections of information were not so accurate and lacking of standard conventions, this might influence the result. Therefore, closed projects are ideal resolutions to validate the proposed models and to enhance their effective contribution in QA activities in order to improve the software development process.

APPENDIX A

In this section, all the metrics that are supported by EvAn tool are introduced. These metrics were studied from the book "Object – Oriented Metrics, measures of Complexity" [20][44]

The list of 19 different metrics offered by Metrics View plugin:

No	Metric name	Abbreviation
1	Number of Public Attributes	NOPA
2	Average Method Weight	AMW
3	Number of Methods	NOM
4	Access to Foreign Data – Class level	ATFDClass
5	Lines of Code – Class level	LOCClass
6	Number of Accessor Methods	NOAM
7	Weighted Of a Class	WOC
8	Weighted Method Count	WMC
9	Tight Class Cohesion	TCC
10	Maximum Nesting Level – included average and sum	MAXNESTING_Avg MAXNESTING_Sum
11	Number of Accessed Variables – included average and sum	NOAV_Avg NOAV_Sum
12	Changing Classes	CC_Avg CC_Sum
13	Cyclomatic Complexity	CYCLO_Avg CYCLO_Sum
14	Changing Methods	CM_Avg CM_Sum

Table 19: Metrics Preferences

The following section introduces a shortly description of each metrics amongst different metrics in the Table 15. The descriptions and definitions also use information from the book Object – Oriented Metrics [44] and EvAn tool [20]. These metrics are computed by summing the number of variables (attributes) using or accessing by a method whereas the method is used by other ones, etc.

NOPA – Number of Public Attributes

This metric calculates the number of public attributes of a class

NOPA - Number of Public Attributes						
Definition	The number of public attributes of a class					
Used for	Data Class(88)					
Measured Entity	Class	Definition	Abstract			
		user-defined	-			
Involved Relations						
HAS (contains)	Attribute	Definition	Visibility	Static	Constant	
		measured class	public	-	–	

Figure 15: NOPA

AMW – Average Method Weight

This is the average static complexity of all methods in a class. McCabe's cyclomatic number is used to quantify the method's complexity.

AMW - Average Method Weight							
Definition	The average static complexity of all methods in a class. McCabe's cyclomatic number is used to quantify the method's complexity (Mar02a, McC76)						
Used for	Refused Parent Bequest(145), Tradition Breaker(152)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
HAS	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	+	+	+

Figure 16: AMW

ATFD – Access to Foreign Data

This metric counts the total number of attributes from unrelated classes that are accessed directly or by invoking accessor methods.

ATFD - Access To Foreign Data							
Definition	The number of attributes from unrelated classes that are accessed directly or by invoking accessor methods (Mar02a)						
Used for	God Class(80), Feature Envy(84)						
Measured Entity	Class	Definition	Abstract				
		user-defined	+				
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	–
Involved Relations							
USES (accesses)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class or method	all	+	–	+	–
	Attribute	Definition	Visibility	Static	Const.		
		neither measured class nor a class from the same hierarchy	public	+	–		
USES (calls)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class or method	all	+	–	+	–
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		neither measured class nor a class from the same hierarchy	public	only	–	–	–

Figure 17: ATFD

LOC – Lines of Code

LOC counts the number of lines of code of an operation, including blank lines and comments.

LOC - Lines of Code							
Definition	The number of lines of code of an operation, including blank lines and comments (LK94)						
Used for	Brain Method(92), Brain Class(97)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

Figure 18: LOC

NOM – Number of Methods

This metric counts the total number of defined methods in the selected scope in the source code.

NOAM – Number of Accessor Methods

This is the number of accessor (getter and setter) methods of a class.

NOAM - Number of Accessor Methods							
Definition	The number of accessor (getter and setter) methods of a class						
Used for	Data Class(88)						
Measured Entity	Class	Definition	Abstract				
		user-defined	-				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	public	only	-	-	-

Figure 19: NOAM

WOC – Weight Of a Class

This metric counts the number of "functional" public methods divided by the total number of public members.

WOC - Weight Of a Class							
Definition	The number of "functional" public methods divided by the total number of public members (Mar02a)						
Used for	Data Class(88)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	public	–	–	+	–
	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	public	only	–	–	–
	Attribute	Definition	Visibility	Static	Const.		
		measured class	public	+	–		

Figure 20: WOC

WMC – Weighted Method Count

This is the number of statical complexity of all methods of a class. The CYCLO metric is used to quantify the method's complexity.

WMC - Weighted Method Count							
Definition	The sum of the statical complexity of all methods of a class. The CYCLO metric is used to quantify the method's complexity (CK94, McC76)						
Used for	Refused Parent Bequest(145), Tradition Breaker(152), God Class(80), Data Class(88), Brain Class(97)						
Measured Entity	Class	Definition	Abstract				
		user-defined	–				
Involved Relations							
HAS (contains)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		measured class	all	+	+	+	+

Figure 21: WMC

TCC – Tight Class Cohesion

This metric is the relative number of method pairs of a class that access I common at least one attribute of the measured class.

TCC - Tight Class Cohesion								
Definition	The relative number of method pairs of a class that access in common at least one attribute of the measured class (BK95)							
Used for	God Class(80), Brain Class(97)							
Measured Entity	Class	Definition	Abstract					
		user-defined	–					
Involved Relations								
USES (accesses)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract	
		measured class	all	+	–	+	–	
	Attribute	Definition	Visibility	Static	Const.			
		measured class	all	+	+			

Figure 22: TCC

MAXNESTING – Maximum Nesting Level

The metric indicates the maximum nesting level of control structures within an operation.

MAXNESTING - Maximum Nesting Level							
Definition	The maximum nesting level of control structures within an operation						
Used for	Intensive Coupling(120), Dispersed Coupling(127)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

Figure 23: MAXNESTING

NOAV – Number of Accessed Variables

NOAV counts the total number of variables accessed directly from the measured operation. Variables include parameters, local variables but also instance variables and global variables.

NOAV - Number of Accessed Variables							
Definition	The total number of variables accessed directly from the measured operation. Variables include parameters, local variables, but also instance variables and global variables						
Used for	Brain Method(92)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-
USES (accesses)	Attribute	Definition	Visibility	Static	Const.		
		user-defined	all	+	–		

Figure 24: NOAV

CC – Changing Classes

This metrics counts the number of classes in which the methods that call the measured method are defined.

CC - Changing Classes							
Definition	The number of classes in which the methods that call the measured method are defined in (Mar02a)						
Used for	Shotgun Surgery(133)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	+
Involved Relations							
IS USED (called by)	Class	Definition	Abstract				
		user-defined, scope of operations called from the measured operation (see CM)	+				

Figure 25: CC

CYCLO – McCabe's Cyclomatic Complexity

This is the number of linearly-independent paths through an operation.

CYCLO - McCabe's Cyclomatic Number							
Definition	The number of linearly-independent paths through an operation (McC76)						
Used for	Brain Method(92)						
Measured Entity	Operation	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

CM – Changing Methods

CM indicates the number of distinct methods that call the measured method.

CM - Changing Methods							
Definition	The number of distinct methods that call the measured method (Mar02a)						
Used for	Shotgun Surgery(133)						
Measured Entity	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	+
Involved Relations							
IS USED (is called by)	Method	Definition	Visibility	Get/Set	Constr.	Static	Abstract
		user-defined	all	+	+	+	-

Figure 26: CM

APPENDIX B

In this section, the results' tables in Chapter 6.2 are described.

Table 20: Variables Entered/Removed in Stepwise Regression

Variables Entered/Removed[a]

Model	Variables Entered	Variables Removed	Method
1	ATFDClass	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
2	NOM	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
3	WMC	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
4	NOAV_Avg	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).

5	NOPA		.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
6	WOC		.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
7	CC_Sum		.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
8	NOAM		.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
9	CM_Avg		.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
10		.	WOC	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).

11	CC_Avg	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
12	TCC	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
13	MAXNESTING_Sum	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
14	LOCClass	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
15	NOAV_Sum	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
16	.	MAXNESTING_Sum	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).

17	CM_Sum	.	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).
18	.	CC_Sum	Stepwise (Criteria: Probability-of-F-to-enter <= .050, Probability-of-F-to-remove >= .100).

a. Dependent Variable: defect_density

Table 21: Coefficients in Stepwise Regression

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	.474	.020		23.939	.000
	ATFDClass	.056	.002	.405	23.340	.000
2	(Constant)	.429	.021		20.047	.000
	ATFDClass	.041	.004	.295	11.097	.000
	NOM	.007	.001	.145	5.443	.000
3	(Constant)	.416	.021		19.371	.000
	ATFDClass	.044	.004	.321	11.869	.000
	NOM	.012	.002	.236	7.202	.000
	WMC	-.002	.000	-.139	-4.729	.000
4	(Constant)	.298	.034		8.698	.000
	ATFDClass	.037	.004	.269	9.138	.000
	NOM	.014	.002	.285	8.257	.000
	WMC	-.002	.000	-.163	-5.475	.000
	NOAV_Avg	.032	.007	.086	4.416	.000
5	(Constant)	.297	.034		8.689	.000
	ATFDClass	.036	.004	.261	8.854	.000
	NOM	.014	.002	.276	7.979	.000
	WMC	-.002	.000	-.142	-4.652	.000
	NOAV_Avg	.034	.007	.091	4.656	.000
	NOPA	-.063	.019	-.059	-3.315	.001
6	(Constant)	.349	.040		8.754	.000

	ATFDClass	.036	.004	.264	8.943	.000
	NOM	.013	.002	.269	7.750	.000
	WMC	-.002	.000	-.144	-4.732	.000
	NOAV_Avg	.034	.007	.090	4.619	.000
	NOPA	-.074	.019	-.070	-3.826	.000
	WOC	.061	.024	.045	2.518	.012
	(Constant)	.337	.040		8.415	.000
	ATFDClass	.036	.004	.261	8.860	.000
	NOM	.016	.002	.330	7.879	.000
	WMC	-.002	.000	-.153	-4.988	.000
7	NOAV_Avg	.033	.007	.087	4.480	.000
	NOPA	-.077	.019	-.072	-3.971	.000
	WOC	.064	.024	.047	2.641	.008
	CC_Sum	-.001	.000	-.070	-2.608	.009
	(Constant)	.328	.040		8.181	.000
	ATFDClass	.036	.004	.264	8.966	.000
	NOM	.016	.002	.312	7.393	.000
	WMC	-.002	.000	-.165	-5.361	.000
8	NOAV_Avg	.033	.007	.088	4.538	.000
	NOPA	-.091	.020	-.085	-4.582	.000
	WOC	.044	.025	.033	1.781	.075
	CC_Sum	-.002	.000	-.113	-3.807	.000
	NOAM	.027	.008	.091	3.335	.001
	(Constant)	.275	.042		6.530	.000
	ATFDClass	.037	.004	.269	9.167	.000
	NOM	.016	.002	.322	7.644	.000
	WMC	-.002	.000	-.159	-5.191	.000
	NOAV_Avg	.032	.007	.084	4.324	.000
9	NOPA	-.089	.020	-.084	-4.515	.000
	WOC	.029	.025	.021	1.149	.251
	CC_Sum	-.004	.001	-.221	-5.526	.000
	NOAM	.036	.008	.121	4.274	.000
	CM_Avg	.050	.013	.106	4.013	.000
	(Constant)	.249	.036		6.954	.000
	ATFDClass	.037	.004	.269	9.148	.000
	NOM	.016	.002	.323	7.666	.000
	WMC	-.002	.000	-.159	-5.176	.000
10	NOAV_Avg	.032	.007	.084	4.339	.000
	NOPA	-.085	.019	-.080	-4.379	.000
	CC_Sum	-.004	.001	-.228	-5.761	.000
	NOAM	.038	.008	.129	4.751	.000
	CM_Avg	.052	.012	.110	4.239	.000
11	(Constant)	.277	.036		7.670	.000

	ATFDClass	.038	.004	.277	9.438	.000
	NOM	.015	.002	.296	6.995	.000
	WMC	-.002	.000	-.151	-4.943	.000
	NOAV_Avg	.030	.007	.080	4.120	.000
	NOPA	-.090	.019	-.084	-4.647	.000
	CC_Sum	-.004	.001	-.281	-6.885	.000
	NOAM	.044	.008	.149	5.436	.000
	CM_Avg	.193	.031	.407	6.277	.000
	CC_Avg	-.233	.047	-.274	-4.991	.000
	(Constant)	.262	.037		7.131	.000
	ATFDClass	.038	.004	.272	9.278	.000
	NOM	.015	.002	.297	7.030	.000
	WMC	-.002	.000	-.153	-5.006	.000
	NOAV_Avg	.035	.008	.094	4.602	.000
12	NOPA	-.084	.020	-.079	-4.296	.000
	CC_Sum	-.005	.001	-.292	-7.098	.000
	NOAM	.048	.008	.161	5.760	.000
	CM_Avg	.198	.031	.416	6.403	.000
	CC_Avg	-.241	.047	-.283	-5.144	.000
	TCC	-.075	.034	-.041	-2.190	.029
	(Constant)	.233	.039		5.969	.000
	ATFDClass	.041	.004	.297	9.445	.000
	NOM	.018	.003	.367	6.934	.000
	WMC	-.002	.000	-.124	-3.692	.000
	NOAV_Avg	.038	.008	.103	4.939	.000
	NOPA	-.081	.020	-.076	-4.152	.000
13	CC_Sum	-.005	.001	-.304	-7.331	.000
	NOAM	.047	.008	.160	5.726	.000
	CM_Avg	.191	.031	.403	6.176	.000
	CC_Avg	-.229	.047	-.269	-4.863	.000
	TCC	-.082	.035	-.045	-2.384	.017
	MAXNESTING_Sum	-.005	.002	-.114	-2.186	.029
	(Constant)	.249	.039		6.302	.000
	ATFDClass	.038	.004	.275	8.458	.000
	NOM	.016	.003	.318	5.664	.000
	WMC	-.002	.000	-.164	-4.457	.000
	NOAV_Avg	.037	.008	.099	4.754	.000
14	NOPA	-.079	.020	-.074	-4.013	.000
	CC_Sum	-.005	.001	-.288	-6.870	.000
	NOAM	.054	.009	.183	6.261	.000
	CM_Avg	.183	.031	.385	5.877	.000
	CC_Avg	-.209	.048	-.246	-4.399	.000
	TCC	-.079	.035	-.043	-2.282	.023

	MAXNESTING_Sum	-.009	.003	-.199	-3.249	.001
	LOCClass	.000	.000	.169	2.634	.008
	(Constant)	.215	.041		5.188	.000
	ATFDClass	.041	.005	.299	8.883	.000
	NOM	.021	.003	.419	6.226	.000
	WMC	-.002	.001	-.181	-4.859	.000
	NOAV_Avg	.047	.009	.125	5.459	.000
	NOPA	-.078	.020	-.073	-3.981	.000
15	CC_Sum	-.005	.001	-.304	-7.193	.000
	NOAM	.058	.009	.198	6.654	.000
	CM_Avg	.181	.031	.381	5.826	.000
	CC_Avg	-.206	.048	-.242	-4.325	.000
	TCC	-.072	.035	-.039	-2.067	.039
	MAXNESTING_Sum	-.003	.004	-.074	-.965	.335
	LOCClass	.001	.000	.296	3.731	.000
	NOAV_Sum	-.003	.001	-.356	-2.717	.007
	(Constant)	.214	.041		5.165	.000
	ATFDClass	.041	.005	.300	8.917	.000
	NOM	.021	.003	.427	6.406	.000
	WMC	-.003	.000	-.187	-5.108	.000
	NOAV_Avg	.048	.008	.128	5.682	.000
	NOPA	-.079	.020	-.074	-4.042	.000
16	CC_Sum	-.005	.001	-.306	-7.261	.000
	NOAM	.058	.009	.197	6.647	.000
	CM_Avg	.184	.031	.387	5.942	.000
	CC_Avg	-.211	.047	-.249	-4.481	.000
	TCC	-.068	.034	-.037	-1.986	.047
	LOCClass	.001	.000	.297	3.744	.000
	NOAV_Sum	-.004	.001	-.433	-4.127	.000
	(Constant)	.224	.042		5.378	.000
	ATFDClass	.041	.005	.299	8.905	.000
	NOM	.021	.003	.411	6.123	.000
	WMC	-.003	.000	-.193	-5.245	.000
	NOAV_Avg	.047	.008	.126	5.597	.000
	NOPA	-.081	.020	-.076	-4.151	.000
17	CC_Sum	.001	.003	.055	.293	.770
	NOAM	.056	.009	.191	6.404	.000
	CM_Avg	.213	.034	.447	6.224	.000
	CC_Avg	-.282	.059	-.332	-4.769	.000
	TCC	-.061	.035	-.033	-1.747	.081
	LOCClass	.001	.000	.345	4.161	.000
	NOAV_Sum	-.004	.001	-.464	-4.380	.000
	CM_Sum	-.002	.001	-.345	-1.981	.048

	(Constant)	.222	.041		5.424	.000
	ATFDClass	.041	.005	.300	8.925	.000
	NOM	.021	.003	.415	6.311	.000
	WMC	-.003	.000	-.192	-5.240	.000
	NOAV_Avg	.048	.008	.127	5.621	.000
	NOPA	-.081	.020	-.076	-4.144	.000
18	NOAM	.057	.009	.193	6.596	.000
	CM_Avg	.209	.032	.441	6.464	.000
	CC_Avg	-.273	.050	-.321	-5.492	.000
	TCC	-.062	.034	-.034	-1.806	.071
	LOCClass	.001	.000	.338	4.262	.000
	NOAV_Sum	-.004	.001	-.461	-4.377	.000
	CM_Sum	-.002	.000	-.295	-7.525	.000

a. Dependent Variable: defect_density

REFERENCE LIST

[1] M. Jureczko, and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction", Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE'10, ACM, 2010.

[2] T. Illes-Seifert, and B. Paech, "Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs", Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic & Industrial Conference, Information and Software Technology, vol. 52, 2010, pp. 539 – 558.

[3] J. Grundy, "Aspect-oriented requirements engineering for component based software systems", Proceeding of 4th International symposium on Requirement Engineering (Limerick, Ireland, 7-11 June 1999), RE 99, IEEE Computer Society, Washington, DC, 1999.

[4] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", Future of Software Engineering, 2007. FOSE '07, 23-25 May 2007, pp. 85 – 103.

[5] S. Lessmann, C. Mues and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings", IEEE Transaction on Software Engineering, vol.34, no.4. July/August 2008.

[6] B. Caglayan, A. Tosun, A. Miranskyy, A. Bener and N. Ruffolo, "Usage of Multiple Prediction Models Based on Defect Categories". Bogazici University Department of Computer Engineering. IBM Canada Ltd..

[7] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors", Software Engineering, IEEE Transactions on, 33(1):2–13–, 2007.

[8] B. Turhan, and A. Bener, "A multivariate analysis of static code attributes for defect prediction", In Quality Software, 2007. QSIC '07. Seventh International Conference on, pp. 231-237.

[9] A.G. Koru, and K.E. Emam, "The theory of relative dependency: Higher coupling concentration in smaller modules", IEEE Software, 27:81-89, 2010.

[10] D. Wahuydin, A. Schatten, D. Winkler, A. M. Tjoa, S. Biffl, "Defect Prediction using Combined Product and Project Metrics A Case Study from Open Source "Apache" My Faces Project Family", Institute for Software Technology and Interactive Systems, Vienna University of Technology, Vienna, Austria; Software Engineering and Advanced Applications, pp. 207 – 215, 2008.

[11] S.R. Chidamber, and C.F. Kemerer, "A metrics suite for object oriented design", IEEE Transaction on Software Engineering, 20, 6, DOI=http://doi.ieeecomputersociety.org/10.1109/32.295895, June 1994, pp. 473 – 493

[12] B. Henderson-Sellers, “Object-Oriented Metrics, measures of Complexity”. Prentice Hall, 1996.

[13] T.J. McCabe, “A complexity measure”, IEEE Trans, On Software Engineering, 2, 4, pp. 308-320, 1976.

[14] M.D.M Suffian, and M.R. Abdullah, “Establishing a defect prediction model using a combination of product metrics as predictors via Six Sigma Methodology”, Information Technology (ITSim), 2010 International Symposium in, Kuala Lumpur, Malaysia, pp. 1087 – 1092, 2010

[15] Six Sigma, Available at http://www.sixsigmaonline.org/index.html

[16] N.E. Fenton, and M. Neil, “A Critique of Software defect prediction model”, Software Engineering, IEEE Transactions on, vol. 25, issue 5, pp. 675 – 689, 1999.

[17] Charles River Analytics, “About Bayesian Belief Network”, Charles River Analytics, Inc 2004, available at https://www.cra.com/pdf/BNetBuilderBackground.pdf, last revisited 27.05.2011.

[18] S. Kietzmann, and N. Foster, “What is Quality Assurance?”, Conjecture Corporation, 2003 – 2011, last modified date: 18 May 2011.

[19] J. Mangino, “IPCC Good Practice Guidance and Uncertainty Management in National Greenhouse Gas Inventories. Chapter 8: Qualuty Assurance and Quality Control”, Cross-sectoral Methodologies for Uncertainly Estimation and Inventory Quality, Expert-Group: Quality Assurance and Quality Control (QA/QC).

[20] S.M. Olbrich, “Evolution of Software Systems: A Study of the Maintainability of God Classes and Brain Classes in Object-Oriented Software Systems”, Department of Computer Science, University of Applied Sciences Mannheim, Germany, Chapter 3: Software Solution: The EvolutionAnalyzer, 2010.

[21] R. Moser, W. Pedrycz and G.Succi, “A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction”, Free University of Bolzano-Bozen, Italy, University of Alberta, Canada, ICSE ’08 Proceedings of the 30th international conference on Software Engineering, 2008.

[22] M. Klaes, H. Nakao, F. Elberzhager and J. Muench, “Predicting Defect Content and Quality Assurance Effectiveness by Combining Expert Judgment and Defect Data – A Case Study”, Fraunhofer Institut Experimentelles Software Engineering, IESE-Report No. 064.08/E, 2008.

[23] N.E. Schneidewind, “Body of Knowledge for Software Quality Measurement”, Dept. of Inf. Sci, Naval Postgraduate Sch., Monterey, CA, IEEE Computer Society, 2002.

[24] D. Galin, “Software quality assurance: from theory to implementation”, Pearson Education, 2004.

[25] M. R. Lyu, “Handbook of Software Reliability Engineering”, IEEE Computer Society Press, McGraw Hill Book Company, 1995.

[26] N. E. Schneidewind, “Analysis of Error Processes in Computer Software”, Sigplan Note, vol. 10, no. 6, pp. 337-346, 1975.

[27] J. D. Musa, K. Okumoto, ”Software Reliability Models: Concepts, Classification, Comparisons and Practice”, Electronic Systems Effectiveness and Life Cycle Costing, J.K. Skwirzynski (ed.) , NATO ASI Series, F3, Springer-Verlag, Heidelberg, pp. 395-424.

[28] S. Biffl, W. J. GutJahr, “Using a Reliability Growth Model to Control Software Inspection”, Research Group Industrial Software Engineering, Institute for Software Technology and Interactive Systems, Vienna University, Vienna, Austria; Empirical Software Engineering, 7, pp. 257-284, 2002.

[29] A. Wood, “Software Reliability Growth Models”, Tandem Computers, Tandem Technical Report 96.1, Part Number: 130056, 1996.

[30] P. L. Li, J. Herbsleb, M. Shaw, “Finding Predictors of Field Defects for Open Source Software Systems in Commonly Available Data Sources: a Case Study of OpenBSD”, Institute for Software Research, Internation School of Computer Science, Carnegie Mellon University; 11th IEEE International Software Metrics Symposium, 2005.

[31] P. L. Li, J. Herbsleb, M. Shaw, “Forecasting Field Defect Rates Using a Combined Time-based approach and Metric-based approach. A case study of OpenBSD”, Institute for Software Research, Internation School of Computer Science, Carnegie Mellon University; Proc. ISSRE, 2005.

[32] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnel, and A. Wesslen, “Experimentation in software engineering: An Introduction”, International Series in Software Engineering, Vol. 6, Springer, pp. 228, 2000.

[33] P. L. Li, M. Shaw, J. Herbsleb and B. Robinson, “Experiences and results from initiating field defect prediction and product test prioritization efforts at ABB Inc”, Proceeding of the 28th ICSE, Shanghai, China: ACM, 2006.

[34] “IEEE standard for a software quality metrics methodology”, Software Engineering Standards Committee of the IEEE Computer Society, IEEE-SA Standards Board, 1998.

[35] T. Gyimothy, “Software product metrics”, Universitas Scientiarum Szegediensis, University of Szeged, Department of Software Engineering,

[36] S. Purao and V. Vaishnavi, “Product Metrics for Object-Oriented Systems”, ACM Computing Surveys (CSUR), Volume 35, Issue 2, 2003.

[37] G. Denaro and M. Pezze, “An Empirical Evaluation of Fault-Proneness Models”, Proceedings of the 7th International Conference on Software Engineering (ICSE2002), Miami, USA, 2002.

[38] T. M. Khoshgoftaar, R. Shan and E. B. Ellen, “Using product, process, and execution metrics to predict fault-prone software modules with classification trees”, High Assurance Systems Engineering, 5th IEEE International Symposim on HASE, pp. 301-310, 2002.

[39] G. J. Evans, “A Framework for Project Metrics: Deciding what to measure and how to measure it”, CVR/IT Consulting LLC, www.cvr-it.com, 2007.

[40] "A Guide to the Project Management Body of Knowledge (PMBOK Guide)", Project Management Institute, American National Standards Institute (ANSI), 2008.

[41] E. J. Weyuker, T. J. Ostrand and R. M. Bell, "Using Developer Information as a Factor for Fault Prediction", Proceeding of the Third International Workshop on Predictor Models in Software Engineering, IEEE Computer Society, 2007.

[42] A. Mockus, P. Zhang and P. L. Li, "Drivers for Customer Perceived Software Quality", International Conference on Software Engineering (ICSE), Saint Louis, 2005.

[43] P. L. Li, "Metrics based field problem prediction", Metrics Lesson Fianl, ISRI – SE – CMU.

[44] B. Henderson-Sellers, "Object-Oriented Metrics, measures of Complexity", Prentice Hall, 1996.

[45] Project Analyzer Help, "Complexity metrics", Aivosto Oy, available at www.aivosto.com.

[46] E. Norling and S. Swartz, "Literature – Where and How to search", Blekinge Institute of Technolgy, Ronneby, Sweden.

[47] Project Analyzer Help, "Lines of code metrics (LOC)", Aivosto Oy, available at www.aivosto.com.

[48] School of Geography, "Stepwise Linear Regression", "School of Geography, University of Leeds".

[49] Wikipedia, "Stepwise Regression", Wikipedia The Free Encyclopedia, available at http://en.wikipedia.org/wiki/Stepwise_regression, last retrieved on 19.12.2011.

[50] Absolute Astronomy, "Stepwise regression", available at http://www.absoluteastronomy.com/topics/Stepwise_regression, last retrieved on 19.12.2011.

[51] J Kaur, S. Singh, and K. S. Kahlon, "Comparative Analysis of the Software Effort Estimation Models", World Academy of Sciences, Engineering and Technology 46, 2008.