CG0174 MSC COMPUTING PROJECT

DISSERTATION 2009/2010

Investigation in the automatic assessment of software's design quality. Development of a tool to support the automatic assessment of code using Object-Oriented metrics.

06009979 Demetris Siampanias

SCHOOL OF CEIS

MSC COMPUTER SCIENCE

northumbriaUNIVERSITY

DECLARATION

I declare the following:

That the material contained in this dissertation is the end result of my own work and that due acknowledgement has been given in the bibliography and references to all sources be they printed, electronic or personal.

The Word Count of this Dissertation is *23,500*

That unless this dissertation has been confirmed as confidential, I agree to an entire electronic copy or sections of the dissertation to being placed on Blackboard, if deemed appropriate, to allow future students the opportunity to see examples of past dissertations. I understand that if displayed on Blackboard it would be made available for no longer than five years and that, students would be able to print off copies or download. The authorship would remain anonymous.

I agree to my dissertation being submitted to a plagiarism detection service, where it will be stored in a database and compared against work submitted from this or any other School or from other institutions using the service. In the event of the service detecting a high degree of similarity between content within the service this will be reported back to my supervisor and second marker, who may decide to undertake further investigation that may ultimately lead to disciplinary actions, should instances of plagiarism be detected.

I have read the UNN/CEIS Policy Statement on Ethics in Research and Consultancy and I confirm that ethical issues have been considered, evaluated and appropriately addressed in this research. Ethics Form B is attached at the back of the dissertation.

SIGNATURE: ……. DATE: …./…./….

ABSTRACT

In parallel with the rise of prominence of the object-oriented paradigm has come the acceptance that software metrics for measuring object-oriented systems. Several software practitioners and academics have developed metrics suited to the OO paradigm. Despite the large acceptance of this paradigm, the design principles and the intimate mechanisms of object-orientation are not clearly understood and realised by students in University. The issue arises when Tutors manually assess the quality of such object-oriented systems as it is often difficult to measure the design quality e.g. degree of code reuse, complexity, maintainability or flexibility, etc., because of the number and size of classes but also due to the associated issues in manual assessment. Automatic analysis of code has been proven effective enough for tackling the issues associated with manual marking in combination with software metrics.

This dissertation lists existing metrics that have been developed as appeared in the literature and a suite has been constructed with metrics that deemed as more appropriate based on a classification of the object-oriented concepts and mechanisms for marking students' assignments. This derived in the development of a system that allows the dynamic addition of metrics for measuring the object-oriented quality of the code.. This document describes how the system has been designed, implemented and tested. The results of the testing revealed that using metrics the quality of the code can be measured successfully but would not be reliable enough with the current specifications of the metrics as defined in the literature. Therefore suggestions have been made on how these issues can be solved for the system to be applicable for its purpose.

CONTENTS

APPENDICES

FIGURES

INTRODUCTION

The fundamental purpose of this project is the depicted on the two following aims

"identify, categorise and choose Object-Oriented Metrics that could be utilised for assessing the design quality of students programming assignments"

"design, implement, test an automatic assessment tool that can use Object-Oriented metrics for evaluating Java code's design for specified design quality attributes which can be used to provide informative feedback to the user."

Which have been extracted as expressed in the hypothesis that:

"The development of a system, that will utilise metrics for automatically assessing the code quality, could provide accurate formative feedback which will reflect on the code's design quality."

BACKGROUND

In the last decade the object-oriented paradigm has decisively influenced the world of software engineering. On the other hand, in spite of the large acceptance of this paradigm, the design principles and the intimate mechanisms of object-orientation are not clearly understand and realised, and this fact has an outcome the development of poorly designed, difficult to understand and complicated object-oriented systems. Main issue that arises is when manually assessing the quality of such object-oriented systems by Tutors at Universities as it is often difficult to measure the design quality and how well the object-oriented concepts have been realised and applied. Automatic assessment has been proven to improve the issues associated with manual assessment.

This document considers the construction, of a system that will be developed to automatically provide information and formative feedback to the users to evaluate the design of the students' software, in order for the Tutors to attain feedback on how well the software is designed and implemented. Software metrics have been widely acknowledged as a successful way for measuring design quality. Despite that, different projects have different levels of required design qualities in the code. Thus possible metrics will be investigated and selected from the ones practitioners and academics have developed.

OBJECTIVES

The objectives, the sub-goals needed to be achieved for completing the project that have been evolved around the aims of the project are as follow:

- To review the problems involved when manually assessing software's quality and how automatic assessment would resolve those
- To review how using metrics could be used in assessing the software's quality

- To identify and review the metrics available for automated assessment
- To select criteria and evaluate the metrics identified
- To design and implement a tool that will automatically assess Java classes based on the selection of required metrics. The tool will provide an overall mark which indicates the design quality of the class being assessed.
- To test the project's hypothesis
- To evaluate the project and the project's success

WORK DONE AND RESULTS

The system has been successfully designed, implemented and tested using an iterative methodology composed of phases, where for each phase one specific requirement of the system has been developed. All the requirements specified have been successfully implemented. An appropriate set of metrics have been constructed suited for marking student assignments which have been incorporated in the system. From the hypothesis testing the quality of the feedback gained from the system has been analysed. The results showed that the in total the system provided very good results but it was not without its problems, since some metrics have been identified that can be inaccurate which was an issue that concerned their specification and appropriate changes have been suggested for enhancing accurateness. Another important conclusion was that, to attain reliable results the system has to be configured as appropriate. The system provides this flexibility of adjusting the boundaries so the tutors can raise and lower the boundaries as appropriate to attain the desirable results.

STRUCTURE OF THE REPORT

The report has been structured to include the following chapters:

LITERATURE REVIEW

MEASURING SOFTWARE QUALITY

Describes the complications associated with measuring software quality and the object-oriented concepts and introduces the concept of automatic assessment.

SOFTWARE METRICS

Introduces software metrics, both object-oriented and traditional and based on an overall survey in the literature. Metric suites are introduced and evaluated.

EVALUATION AND SELECTION OF METRICS

Classifies and categorises metrics for appropriate ones to be chosen upon the object-oriented concepts they measure. The list of chosen metrics is presented.

PRACTICAL WORK

The practical work done for designing, implementing and testing the system and the specification of the system's requirements.

HYPOTHESIS TEST

Describes the experiment performed for testing the hypothesis and lists the results of the testing. Recommendations are made for improving the results as identified from the testing.

EVALUATION

Thorough and critical evaluation of the system and the process of developing the system.

LITERATURE REVIEW

MEASURING SOFTWARE QUALITY

This chapter presents the problem in hand by demonstrating the problems that arise when assessing software's design quality through a description of the object-oriented concepts used for expressing this quality. This problem arose from the problems Tutors have when manually assessing the code quality for student programming assignments and this chapter proposes a solution by automating this assessment process. Concluding the process automation has been identified that can be achieved using Object-Oriented metrics, that have been widely acknowledged as a successful way for measuring software design quality.

1.1 SOFTWARE DESIGN

"Design is the backbone of any software system"

Design's importance as denoted by the statement of Abreu (2001), lies in the fact that without an effective "backbone", growth and change is very difficult if not incapable to achieve. Thus the development of any system requires the implementation of systems which are easy to understand, amend, and test. *Design is intermediately linked with quality*. Well designed systems have good quality and therefore badly designed systems lack quality. Understanding thus the Object-Orientation concepts and features is important in developing well designed systems with good quality. The Object-Oriented paradigm offers mechanisms such as abstraction, message passing, coupling, cohesion and inheritance, to accommodate the development of large and controllable systems. In By contrast employing conventional methods for software development is much more different than Object-Oriented design (Li and Henry, 1993). In the recent past years the adoption of the object-oriented paradigm has influenced the area of software development and engineering. On the other hand, in spite of the large acceptance of this paradigm, the design principles and the concept of object-orientated design/programming are not clearly understand and realised, and this lead to the outcome of poorly developed systems, difficult to understand and complicated object-oriented systems. Even in Academic institutions, as shown on a survey by Clua and Feldgen (2008), students when writing small programs, have problems in realising and applying the Object-Oriented concepts and more specifically in applying abstraction and encapsulation and generally developing well designed systems.

1.2 OBJECT-ORIENTED PARADIGM

In the recent years there has been a large acceptance of the Object-Oriented paradigm (Amandeep et al., 2009). The employment of object-oriented design has as main objective in facilitating the development of systems that are more maintainable, flexible and extensive by reducing the complexity of software at both the system and class level (McConnell, 2004). Systems that have been designed successfully using Object-Oriented design are flexible, mainly because they manage the dependencies of interclass's. Understanding the concepts object-oriented is the initial step towards the selection of metrics for evaluating the Object-Oriented concept. The object-oriented design includes concepts such as object, class, attributes, inheritance, method, message passing, coupling, cohesion, polymorphism, encapsulation and abstraction (Li and Henry, 1993). Booch (1994) gives a very indicative definition of what Object Oriented programming, and states:

"Object-oriented programming is a method of implementation in which programs are organised as cooperative collections of objects, each of which represents an instance of some class, and whose class are all members of a hierarchy of classes united via inheritance relationships."

1.2.1 OBJECT-ORIENTED PROGRAMMING FUNDAMENTAL CONCEPTS

The most important concepts of Object-Oriented programming as stated by Bennett et al. (2006) include class, object, instance, generalisation, message passing and polymorphism. These six fundamental concepts are listed and specified below:

- *Class (Methods & Attributes) & Objects and Instances,* a class describes a set of objects that are specified in the same way. *Instance* mainly defines a single *Object*, which might represent a single person, a thing or concept in the application domain.
- *Generalisation,* describes the creation of general objects higher in the hierarchy (superclass) that will encapsulate certain common characteristics for the lower objects in the hierarchy (class) to inherit.
- *Message passing,* characterises the exchange of messages in an object-oriented system between objects for interacting with each other for requesting operations or attainment of information.
- *Polymorphism,* describes the ability of different objects to realise certain features in multiple different ways.

1.2.2 OBJECT-ORIENTED DESIGN PRINCIPLES AND MECHANISMS

These fundamental concepts of Object-Oriented Programming offer features for enhancing the code quality and improving the system's design. Such include coupling, cohesion, inheritance and encapsulation.

- *Coupling* describes the design components interconnection extent, and is viewed by the total associations that an object with other objects and its interactions with other objects. Coupling can be divided into two distinct types that a class can have, *afferent* where it's the number of other classes using a class and *efferent* when a class is used by other classes.
- *Cohesion* is the measure that describes the degree to which an object is built for a single purpose. In contrast with coupling that measures the degree that classes interact with each

other, cohesion focuses on how single class is designed. Cohesion has three types, operation class and specialisation cohesion (Bennett et al., 2006).

- *Inheritance* is the mechanism for implementing the generalisation and specialisation concepts in object-oriented programming languages. Inheritance helps in keeping programs shorter and more tightly organised by declaring certain specifications that can be shared by multiple parts of the program.
- *Encapsulation* of objects is achieved by message passing concept, as it allows object to encapsulate their internal information from other elements. Therefore, each of the elements can be modified and maintained without influencing the rest of the elements in the system.

Figure below depicts the structural concepts of the Object-Oriented paradigm. As shown a class can have attributes and methods, of which methods can be inherited by a child class, Class B. The classes are used as parents when creating the objects (Object1). The diagram depicts an inheritance tree where the Objects Inherit for parent classes (Class B, C) and these inherit from Class A. Object1 classes Object2 exchange messages and this is when coupling occurs as the message passing couples classes. Polymorphism cannot be seen at this point but generally each Classes B, C, D can realise their methods inherited by Class A with different ways.

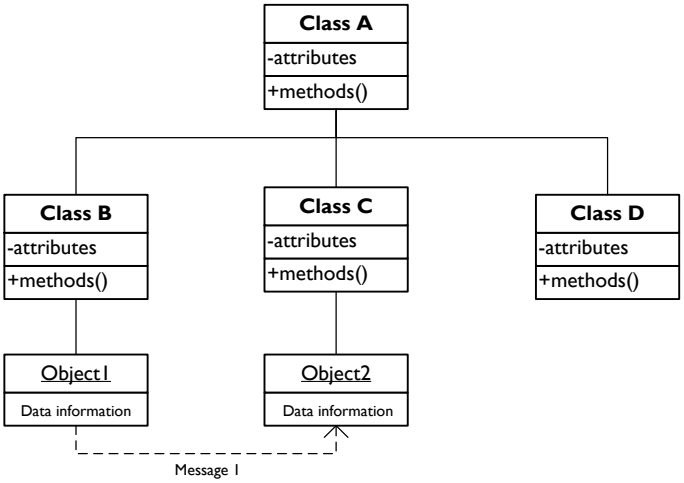


FIGURE 1: OBJECT-ORIENTED PARADIGM STRUCTURAL CONCEPTS

These design principles and mechanisms are offered by the fundamental concepts that Object-Oriented programming has to offer and contribute to the development of well designed systems and are being taught by students during the undergraduate and postgraduate levels. A high level of cohesion and low level of coupling and generally the application of these mechanisms within a system of modules in turn results in a number of overlapping system properties (Karimi and Konsynski, 1988) like reduction of the system's complexity, maintainability, understandability, reliability, flexibility, effectiveness and even reusability (Bansiya and Davis, 2002). These characteristics are specified in the ISO 9126 (ISO, 2001) quality model which addresses the some of the well known human biases that can affect

adversely the quality of the software under development. Despite that, the appropriate application of these mechanisms is not that easy to be put into practice and attain the benefits that these have to offer to software development and even more complicated is assessing and measuring the quality by evaluating the software's code.

1.3 ISSUES AND WAYS OF MEASURING DESIGN QUALITY

As discussed in the previous section 2.1 software's design is intermediately linked with quality. Well designed systems have good quality and therefore poorly designed systems lack quality. It is though difficult though to measure the design quality based on the design of the software. Measurement is the process by which values are assigned to attributes of entities in the real world so as to describe such entities according to clearly defined rules (Fenton and Pfleeger, 2004). Al-Jaafer and Sabri (2005) highlighted that the importance of automatic software quality assessment lies in two main areas, the industry and education because of the problems involved in the manual measurement of the design's quality.

1.3.1 PROBLEMS ASSOCIATED WITH MANUAL MEASUREMENT

In industry, the evolution of software systems and the development of larger systems often lead in the implementation of inflexible systems that are difficult to maintain because of the non-proper application of the Object-Oriented principles.

In education, the issue that arises is when manually assessing the design quality of such object-oriented systems as it is often difficult to measure the design quality e.g. degree of reusability, complexity, flexibility or maintainability etc., because of the number and size of classes. Temblay et al. (2008) highlights that marking programming coursework has always been a complicated task as it involves many aspects, including testing the program that it provides the required functionality, evaluating the source code to ensure that appropriate coding styles and standards are used (commenting, indentation, programming style, etc.) and the required OO design principles have been put into practice.

Mainly the tasks for Tutors for assessing the programming assignments can be characterised as a long, repetitive and tedious task. This is mainly because markers have to measure as accurately as possible the quality of the code except from the functionality required for the assignments. Especially, when the number of code lines and the number of class increases, mainly for larger student assignments, it introduces more complexity and difficulty to accurately measure precisely the quality of the software in hand. As defined, the assignment's size is limited for programming assignments, thus it is not the main factor of the complication behind assessing programming assignments. The problem that makes assignments hard to assess is the complex requirements of good and correct programming practices, since usually students' code is quite badly written and even seemingly incorrect programs deliver the required functionality.

The marking is mainly based on the application of the Object-Oriented design concepts and principles. Students are required to apply these concepts and in different assignments the students are required to apply only specific concepts of object-oriented programming. Despite that, from the research performed and as also supported by Ala-Mutka (2005), currently set of criteria be used as a basis when marking programming assignments does not exist in the literature. Most typically though, as noted above students are required to display their programming skills by developing small to average sized programs and applying the Object-Oriented concepts. As Ala-Mutka (2005) specifically states, the main requirements being marked are functionality, design and programming style. Moreover there are not widely acknowledged criteria for marking mainly because it's the tutors that decide and emphasize upon which concepts of OO based on their personal experience and the teaching objectives of the modules.

Cheang et al. (2003) list and justify the issues associated with manual marking of programming assignments based on a survey performed on students and teaching staff. Such issues include the judgment of correctness and efficiency, the possible approaches for solving a problem, emphasis on aesthetics, human bias and finally time. The issues listed are rational and these are evident when the marking of assignments takes place.

Marinescu (date) highlights, that the process of identifying problems in the code in a non-automatic manner, is time expensive, unrepeatable and non-scalable. Having that said, it would be supportive if there was a way to automatically provide information and formative feedback to the markers which would represent the effectiveness of the Object-Oriented concepts application and the code's design quality, in order for them to gain an insight on how well their system is designed and the principles established. Cheang et al. (2003) agree by stating that through the automation of the grading of programming assignments a great potential can be reaped.

1.3.2 THE NEED FOR PROCESS AUTOMATION

It's been decades now, and more specifically since the 1960's, were the first researchers such as Hext and Winings (1969), Forsythe and Wirth (1965) and Hollingsworth, (1960) investigated and attempted the development of automation of the assessment process. This was in attempt to automate the evaluation student's programming exercises to improve and consequently simplify the process of marking due to the excessive weekly loads of coursework. Educators have recognised and researches have reported on the practical and pedagogical benefits of automated assessment tools for assessing students' programming coursework (Brusilovsky and Higgins, 2005).

Automatic assessment of software is a concept being investigated in the for a long time now, as revealed, with main purpose to automate the process and ease the assessment of software's design quality for the development of more efficient software with better design quality. Ala-Mutka (2005) survey demonstrates the numerous endeavours made from several authors in

developing automated programming assessment systems and the high popularity gained of these systems in the field of teaching and learning, for assessing, grading, evaluating and managing students programming coursework, by saving the instructors time from assessing the programming assignments manually. Such automated systems offer the capability of handling and evaluating assignments very fast and providing the users immediate feedback for the student's code quality, in just a few seconds. With automated assessment the contents of programming assignments can be parsed automatically and directly analyse the code to identify the quality of the code and whether of the required attributes have been successfully been implemented.

As Moha et al. (2006) discuss, for detecting defects, unused code or errors, tools such as PMD and SmallLint have been developed. Despite that, such tools focus on code-level problems but do not address higher-level software defects. Many others have attempted to develop various frameworks, methods or algorithms for automating the evaluation of software's design, (Karimi and Konsyski, 1988; Ciupke, 1999; Redish and Smith, 1986; Hung et al, 1992) but there approaches again lacked analysis and evaluation of the particular Object-Oriented concepts since the focus is on procedural programming.

Software metrics have been widely acknowledged as a successful way for measuring design quality. For example, Mengel and Ulans (1999), through an automatic collection of software metrics from student's assignments they revealed that metrics can be utilised as clear indicators for portraying students performance and moreover as indicators of needs for instructional development. Amandeep et al. (2009) criticize that the concepts of software metrics are well established and there has been a lot of information in the literature of metrics being established for assessing software quality. This is apparent in the increased progress in the area and the development of such tools which have either being implemented or proposed such as JDepend, JHawk, AUTOMARK++ (Al-Jaafer and Sabri, 2005), etc., for automating the evaluation process and identifying the design quality of the software in place. Most of these tools are not explicitly available in public such as AUTOMARK++ and the ones available require registration and have not been directly implemented for marking.

Despite that, different projects have different levels of required design qualities in the code. Thus this project will consider the utilisation of object-oriented metrics and their employment in a tool which will allow the selection of the required design attributes and the level of analysis depth and will automatically assess the design quality of code and provide both overall and particular feedback of the systems design quality attributes.

HIGHLIGHTS AND CONCLUSIONS

This chapter listed and described the object-oriented concepts and mechanisms for demonstrating the criteria that are used to evaluate and assess object-oriented quality. The issues when manually assessing

these concepts and mechanisms are apparent and therefore the process automation has been acknowledged as a superior way of achieving these results. Researchers investigating the area of automatic code analysis described software metrics as a reliable and effective way for achieving code assessment. Evidently object-oriented quality is of high importance and the process automation would improve the overall marking results and could facilitate tutors in marking and eliminate these issues identified that are associated with manual assessment. Software metrics have been identified as the most appropriate technique to be utilised for measuring Object-Oriented quality. This is because software metrics can provide a quantitative means to control the software development process and the quality of software products. Hence the required results can be gained for measuring students' assignments object-oriented quality. The chapter follows analyses through a literature survey the available metrics developed in the field and reviews possible metrics to attain the require measurements in marking.

SOFTWARE METRICS

"What is not measurable, make measurable"

This statement by Galileo Galilei denotes the importance of quantifying observations in order to gain understanding and insight to identify the source of the problem. The same applies to Computer science as well, thus metrics have been developed to act as instruments for measuring software quality. This chapter gives an overall introduction to software metrics and distinguishes the metrics into two categories, traditional and object-oriented. Metrics Suites are also presented in an attempt to identify appropriate ones to be developed and used in automated marking of student's assignments.

2.1 INTRODUCTION TO SOFTWARE METRICS

The first software metrics were first introduced more than 40 years ago and several other followed up to the recent years. (Xenos, 2006). Ebert and Morschel (1997) define metrics as measures of the software development process and the resulted software and state that software metrics can have a major impact for the analysis and software quality improvement. Metrics are tools that can be used efficiently even after the development of the software.

Metrics can be used for multiple purposes and have a different role in the different situations. Such include their application for analysing source code as an indicator of quality attributes, providing programmers detailed feedback about their program's design quality and also as guidelines for when and where re-factoring is essential. Furthermore metrics have also been used as cost prediction, scheduling activities, and also as a productivity measurement (Clúa and Feldgen, 2008). Emam (2001) states that metrics can also be utilised for quality estimation - as noted above - but also for risk management as well. The fundamental basis, at which the development of object-oriented metrics initiated, was that metrics can be used to predict early in the development faulty classes that could be proven to be un-maintainable (Benlarbi et al., 1999). Apparently utilisation of Object-Oriented metrics can be used for various purposes and is applicable to assist the automatic assessment and evaluation of design.

Object Oriented Metrics have mainly five characteristics, localisation, encapsulation, information, inheritance and object abstraction (Jamali, 2006). Jamali (2006) signified the Object Oriented Metrics classes, where each of them focuses on a certain aspect; Size, Complexity, Coupling, Sufficiency, Completeness, Cohesion, Primitiveness (Simplicity), Similarity and Volatility. Brooks and Buell (1994) justify that software metrics measure two particular levels; the class and system level. At the class level the classes are measured individually. System-level metrics on the other hand deal with the system as a whole and its contained classes. The software that will be developed will be aimed to cover and analyse the system classes and the system as a whole. As an example, evaluating the coupling level of the system all the classes will be taken into account as coupling measures class's dependencies to the other classes by automatically looking into the code and identifying the

associations. Moreover, Emam (2001) defines two types of metrics, static and dynamic, where static are the ones which will be taken into consideration for this system's development as there are only concerned with source code analysis in contrast with dynamic ones where the system has to be executed to be analysed.

2.2 METRICS SURVEYS AND REVIEW

Xenos et al. (2000) survey categorises metrics as *Object-Oriented metrics* made for assesing the object-oriented concepts and *Traditional metrics* for traditional programming which most are also applicable and can be used to evaluate object-oriented software. Xenos et al. (2000), survey provides a comprehensive and extended list (more than 90 metrics) of both traditional and Object-Oriented metrics for which Object-Oriented are futher classified into the Object-Oriented attribute being measured by the metric.The traditional metrics and OO metrics identified by Xenos et al. are presented in Appendix A and are sorted alphabetically. The Object-Oriented metrics are classified into the Object-Oriented concepts and mechanisms; class, method, inheritance, coupling and finally more common metrics. This separation is important as it helps to distinguish between the metrics and their level of measurement as for the marking it will be necessary to have different levels of measurement, class or system level, and furthermore the availability of measuring the different OO attributes.

However, despite that Xenos et al. (2000) survey includes a rather comprehensive list of metrics, it has been discovered that it ommtis rather important metrics which have been introduced in later years (e.g. LCOM1, LCOM2 etc. (Briand et al., no date)) and also some have been ommited such as PF (Polymorphism Factor) from the MOOD metric suite (Brito and Carapuca, 1994). Moreover it has also been identified that the MOOD2 set introruced by Brito in 2001 (Brito, 2001) has not been included.

Another comprehensive survey by Kanmani et al. (2006) listed metrics particularly specified for Object-Oriented programming, separates metrics in inheritance, coupling and cohesion as depicted in Appendix B. Kanmani et al. (2006) survey puts more emphasis in analysing the metrics and the attributes these metrics measure. Specifically the survey reveals that some metrics (eg.SPA, SPD) which are specified for measuring specific attributes like coupling also take into account multi-level inheritance and/or polymorphism. This is important to take into account to ensure that the metrics that will be selected are required to be focused on individual object-oriented attributes.

As identified there is a clear categorisation of the metrics into Traditional and Object-Oriented metrics, therefore in the following section the report looks into some of the most well acknowledged metrics for both catergories.

2.3 TRADITIONAL METRICS OVERVIEW

As it has been shown from Xenos et al. survey, there are a lot of traditional metrics which could be utilised for assessing the system at the class-level which is one of the important for measuring the a

class' complexity. There is an argument among the researches on whether traditional metrics should be used for assessing object-oriented software due to the difference in procedural programming with object-oriented programming. This disagreement has also been identified by Rosenberg and Hyatt (no date). Some authors such as Brooks (1993) argue that traditional metrics aren't very helpful when assessing object-oriented development but others such as McCabe, et al. (1994) support that the traditional evaluation still can be supportive. Halstead metrics are entirely focused on evaluating the complexity of procedural programming. Instead metrics such as Cyclomatic Complexity introduced by McCabe (1976) that measures the control flow complexity of modules based on graph theory. Another common traditional metric which is usually used as a "raw" measure for size in many projects is Source Lines of Code (SLOC) (aka LOC) (Tegarden et al, 1996).

2.4 OBJECT-ORIENTED METRICS OVERVIEW

As it has been shown, from the surveys in the previous section 3.2, numerous metrics have been created by various authors over the past few decades (Abreu and Carapuca, 1994; Benlarbi and Melo, 1999; Briand et al., 1997; Brito and Carapuca 1994; Chidamber and Kemerer, 1994; Li and Henry, 1993; Tang et al., 1999), of which a few of these metrics have undergone some form of empirical validation, and some are actually being used by organisations as part of an effort to manage quality (Emam, 2001). This increased amount of metrics been development is mainly due to the increasing importance being placed on software measurement and quality (Briand et al., 2005). The basic foundation about metrics is that they aid in capturing most of the elements of the object-oriented software complexity.

Now, by looking into the existing research various suites of different mainly Object-Oriented metrics have been put together by various researchers. A set of suites that includes the C.K., MOOD and L.K. metrics suites, will be examined to identify whether the employment of one or more of those could cover the requirements for marking students programming assignments. No specification exact specification is provided as this will be identified further when the final metrics are selected.

2.4.1 MOOSE (C.K.)

Chidamber and Kemerer (1994) developed a suite of language independent and well established metrics based in sound theory which have been one of the most widely adopted and experimentally investigated (Basili et al., 1995) set of metrics. CK metric suite is called MOOSE (Metrics for Object-Oriented Software Engineering) is comprised of the six metrics listed below:

- *Depth of Inheritance Tree (DIT)*
- *Number of Children (NOC)*
- *Coupling between object classes (CBO)*
- *Response for a Class (RFC)*
- *Weighted methods per class (WMC)*
- *Lack of Cohesion of Methods (LCOM)*

2.4.2 MOOD

Furthermore, Brito and Carapuca (1994 r) proposed a suite named Metrics for Object-Oriented Design (MOOD) suite which comprises of the following metrics:

- *Method Inheritance Factor (MIF)*
- *Attribute Inheritance Factor (AIF)*
- *Coupling Factor (CF)*
- *Clustering Factor (CFA)*
- *Polymorphism Factor (PF)*
- *Method Hiding Factor (MHF)*
- *Attribute Hiding Factor (AHF)*
- *Reuse Factor (RF)*

2.4.3 LORENZ AND KIDD (L.K.)

Another suite presented by Lorenz and Kidd is consisted of eleven object-oriented metrics (El-Wakil et al., no date). This suite includes the following metrics:

- *Number of Public Methods (NPM)*
- *Number of Methods (NM)*
- *Number of Public Variables per class (NPV)*
- *Number of Variables per class (NV)*
- *Number of Class Variables (NCV)*
- *Number of Class Methods (NCM)*
- *Number of Methods Inherited (NMI)*
- *Number of Methods Overridden (NMO)*
- *Number of New Methods (NNA)*
- *Average Parameters per Method (APM)*
- *Specialisation Index (SIX)*

HIGHLIGHTS AND CONCLUSIONS

The chapter identified the separation between the software metrics and their classifications in an attempt to have a closer look in the area and what has been achieved up to now. The number of metrics developed is clearly and the list currently in hand is rather extensive. Hence the selection of metrics is clearly not simple and straightforward. Continuing after investigating the available suites composed by various authors all of them have been proven to be incomplete and would not provide the required results. The next chapter evaluates and analyses certain metrics for selecting appropriate ones to be used in evaluating and marking student's assignments.

EVALUATION AND SELECTION OF METRICS

This capter analyses particular metrics as evaluted through the literature by establishing a framework to base the selection upon. Several issues are associated when selecting metrics that have been discovered by authors in the literature and been taken into consideration for avoidance. A general evaluation of the metric suites is presented since the metrics selection will be limited based on those findings and relevant work performed in the field. A framework was contructed based on the concepts and mechanisms of the object-oriented paradigm where certain metrics would be selected and mapped to evaluate each of these having thus a set of metrics that would provide complete evaluation of the code.

When selecting metrics to compose a suite revels the issues associated of having too many metrics to work with and not attaining the results expected for the desired objective (Abreu and Carapuca, 1994). Brito and Carapuca (1994) addressed the issues involved when selecting appropriate metrics and suggested that a framework needs to be adopted for the implementation of metrics' initiatives. Thus, Brito and Carapuca (1994) identified a gradual approach which initiates with the identification of the goals at which the required attributes and quality factors are specified and then these have to be mapped to a numbering system. In this case, the required factors have been identified in the previous chapter from the marking frameworks used in assessment thus it is necessary now based on these frameworks to validate and evaluate which one of the metrics being identified are appropriate to acquire the required results.

3.1 METRIC SUITES ASSESSMENT

In order to narrow down the list of metrics and choose which one should be selected it is important, since already existing validations and evaluations of the metrics exist and repeating the process would be needing far too much time, to look into existing evaluations and collect the results from the literature to verify which metrics would provide results that would be both requisite and apposite.
Sherif and Sanderson (1998) using the metrics by Chidamber and Kemerer (MOOSE) performed an analysis on two different projects. The results showed that this set of metrics can help in identifying the complexity both the system and class level. Also the authors highlight that a class with significantly more methods usually can be defined as being more complex and often can be prone to errors. Chidamber and Kemerer metric suite has proven through the literature that this set of metrics has gain a lot of interest and is currently the most eminent set of object-oriented metrics. Xu et al. (2009) supports that there is no other superior model or algorith than the CK suite.

Comparing the two suites, MOOD and CK both are aimed in evaluating the OO concepts, but the CK one measures the system at the class level, but instead the MOOD metric suite was developed with the aim of addressing and evaluating directly the concepts of Object Oriented Programming (Brito and Carapuca, 1994). Abreu et al. (1995) evaluation of the MOOD metric suite through supporting examples, demonstrated that this suite avoids subjectivity of measurement and thus allows replicability, which more clearly defined means that a different set of values can be retrieved

for the same system. The suites in general do not provide the required attributes for marking but individual metrics might be considered for employment in the tools since various researches identified that particular metrics could produce efficient feedback on the code's quality.

C&K metrics are similar to MOOD metrics and still have some differences. CBO and CF are both metrics for measuring coupling and are rather similar, while CBO provides coupling on the class grade and CF provides coupling on the system grade. C&K metrics and MOOD metrics have both their advantages and disadvantages respectively, C&K metrics focus on source code, and lack in system metrics; MOOD metrics focus on project management, and lack of class metrics. C&K metrics and MOOD metrics focus and are established on different parts of OO software design. On the other hand, they work well together as far as their strength and weakness are concerned (Mao and Jiang, 2008).

Next, the MOOD2 metric suite, the latter version of MOOD, lacks of metrics that can actually be automated, and the remaining metrics are insufficient to cover all the attributes required for covering all aspects when evaluating object-oriented design thus this suite cannot be used standalone. Considering now the LK suite, it lacks of metrics for evaluating the internal structure of the software's design. Primarily, this suite of metrics is especially developed to be applicable to class diagrams (El-Wakil et al., no date). The LK metric suite would be useful for the marking process as it counts the attributes and method of a class. A, large number of attributes or methods in a class, predominantly concludes that the class has not been decomposed successfully and could be further broken down into more classes.

Amandeep et al. (2009) propose a composed set of metrics which is a combination of metrics from various researchers. The set includes twenty-two metrics from the CK suite by Chidamber and Kemerer (1994), and the MOOD suite by Brito and Carapuca (1994) and some metrics by Lee et al. (1995), Briand et al. (1999) and a suite by Henderson-sellers (1996) which is not very acknowledged and very little information is available in the literature. The authors results though the evaluation reach to the conclusion that this framework can be used for a variety of projects and evaluate and compare the performance of the code using the Object-Oriented principles. This set of metrics can also be classified into metrics for evaluating the system both as a whole and at the class level. This is important as the programming assignments size's range from one class to multiple classes, thus there will be cases at which only a single class needs to be evaluated.

Despite what has been said, this set of metrics does not analyse the complexity of the methods and at the class level only the Object-Oriented principles are considered and measurements of the number of methods and attributes in classes. Some of the traditional metrics listed and identified in §2.3 could be utilised and used as additional measures of complexity. Rosenberg (no date) research which included the utilisation of both tradition and object-oriented metrics showed that this combination is most effective. McCabe's (1976) metric of Cyclomatic complexity has been acknowledged and widely used since it can effectively measure the complexity of modules and also because it is easy to calculate (Tegarden et al, 1996). Xu et al. (2009) in the reseach suggested that

SLOC (section 3.3) should be continued to be used as it can be a usefull indicator of the size even in Object-Oriented programs.

From this chapter it has been observed that despite that several suites already exist without its flows as identified. Therefore it was not possible to rely on the selection of a single suite to achieve the required results and also Traditional metrics such as Cyclomatic Complexity is not included in any of the suites. The analysis therefore will continue on in looking related work done in this field as discussed in the next chapter.

3.2 RELATED WORK

Another important area to look into for selecting a set of metrics is what metrics are used in similar projects. Brooks and Buell (1994) for a similar project developed an automated tool for automatically gathering the metric values. For this similarly the metrics were categorised into class level and system level and the selected metrics included the metrics from the CK suite. Xie et al. (2000) employed as well McCabe's Cyclomatic Complexity metric.

An identical research by Al-Jaafer and Sabri (2005) for the development of an automated tool that uses metrics for assessing Java programs employed the MOOD metric suite. The authors reached to the conclusions that this suite is the most suitable for assessing object-oriented programs and in particular Java assignments, but also is able to find the deficiencies in the programs under assessment. Despite that they have reached to another important conclusion that this suite is not effective for small programs. Al-Jaafer and Sabri (2004; 2005) successfully in two later projects instead of the MOOD suite, they employed a combination of CK and LK metric suites. CK is well acknowledged and as became apparent from the literature it is a well established suite which can give insights on program's design quality. The choice over the LK suite was made mainly because the suite is composed of class and operation orientated metrics. By simply counting the number of methods and attributes in a class it can be estimated that a high number of methods and attributes implies that the class could be further been decomposed and be broken down and decomposed further.

Evidently, taking a decision of which metrics should be used is not an easy choice and this is more complicated since – as discussed in previously in §1.3 – there is not an established framework form marking thus the selection of metrics needs to be based on the known frameworks and the teaching objectives of programming in academic institutions.

Despite the large number of metrics identified, the most explored metrics which have also been mostly utilised and evaluated are CK and the MOOD metric suites and other individual one identified in the previous section, such as McCabe's Cyclomatic Complexity. One that is not that widely acknowledged but it became apparent from Al-Jaafer and Sabri (2004; 2005) research is the LK suite which can be utilised in the particular case for evaluating students' programming code.

3.3 CLASSIFICATION AND SELECTION OF METRICS

A consequential approach for the classification of object-oriented metrics has been established that was initialised by looking into the fundamental observable concepts of object-oriented paradigm. The OO metrics can be split into two main categories, the system and the class level. These can be further broken down and associated with the attributes are used in teaching in universities discussed in §1.2.1 & §1.2.2. These categories have been classified as shown in FIGURE 2. This categorisation was main objective is to identify the most appropriate metrics for each of the categories. These categorisations focus on identifying metrics which will evaluate the specified attributes of the system and would help gain sufficient results on the quality of the code. The section discusses the categorisation and which metrics were found to be more specific and helpful to be used in measuring each certain aspect. The selection relied in that at least one metric will be selected for each of the categories in order to attain a complete view of the system and its overall object-oriented design quality.

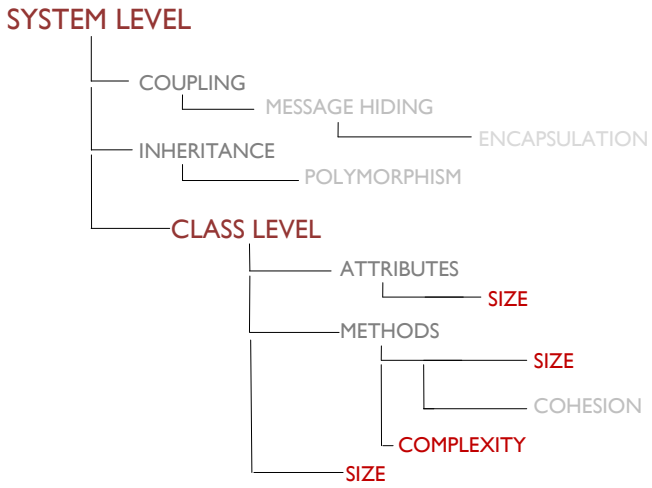


FIGURE 2: PROGRAM EVALUATION HIERARCHY

3.3.1 SYSTEM LEVEL

At the system level there are two main attributes for measurement, coupling and inheritance. Proceeding lower in the hierarchy of the system level, a system is composed of classes, and the classes further define both Attributes and Methods.

COUPLING

A class is coupled with one more classes if the methods of one class use the methods or attributes of the other classes. Mainly high level of coupling is causes issues in maintaining the class because of its dependencies. Such metrics for measuring coupling is CBO (Coupling between Object Classes), which denotes with to how many other classes a class is associated with. CBO therefore could be a very good indicator to identify where the class is loosely coupled and therefore how good the design of the system overall is and the reusability of the class (Chidamber and Kemerer 1994; Basili, et al. 1996). Moreover high CBO signifies that a class is difficult to maintain because of it is overly dependent on other classes.

Another metric for measuring Coupling is RFC (Response For Class). This measures the complexity of the class in terms of method calls. More specifically the number of methods invoked when a method is invoked. Chidamber and Kemerer (1994) and Basili, et al. (1996) justified that when too many methods are invoked makes the class more difficult to test and therefore a low RFC is more desirable.

MESSAGE HIDING/ENCAPSULATION

Information Hiding is the concept that is providing by the encapsulation mechanism of Object-Oriented programming. Such metrics for analysing this concept are the AHF and MHF metrics as included in the MOOD metric suite (see §2.4.2). Al-Jaafer and Sabri (2005) analysis of the Mood suite and particularly these to metrics showed that these metrics are very good quality indicators for Object-Oriented software. High MHF can reveal insufficient abstracted implementation of classes and high AHF can indicate the complexity of the program (Abreu and Carapuca 1994b).

INHERITANCE

At the system-level classes are related with associations. Inheritance is the association where a class uses a class's characteristics as implemented by a parent class. NOC (Number Of Children) determines the number of sub-classes that a parent class has. DIT (Depth of Inheritance Tree) on the other hand measures the depth of inheritance of a class within the hierarchy tree. The deeper a class is in the hierarchy the more complex it is to understand the behaviour of the class therefore low DIT is more desirable. (Chidamber and Kemerer, 1994)
Polymorphism will not be considered at this point as it is an aspect of Object-Oriented programming that will be very difficult to measure using static analysis.

3.3.2 CLASS LEVEL

The class as noted (see §1.2.1) is one of the fundamental concepts of the object-oriented paradigm. At this level is important to identify metrics for both measuring the class aspects, and identifying the classes' connections with other classes along with the interconnected objects of the class. Metrics also have been developed suited for measuring a class's size. LOC for example could be employed for measuring the class's size as larger classes tend to be more difficult to read, maintain and therefore reuse.

ATTRIBUTE

Attributes define the properties of data object in a class. A class with too many defined attributes most commonly results in a class that is low in cohesion and does not promote flexibility. Therefore counting the number of attributes in a class could provide a smart indicator for identifying whether or not a class lacks of quality. Such metrics are included in Lorenz and Kidd metric suite (see §2.4.3).

METHOD

The methods in a class implement its functionality. However, the number of methods in a class affects the overall a class's quality since a class with too many methods usually implies that the class could be further decomposed. Lorenz and Kid suite as identified in the previous chapter includes such metrics for measuring the size of methods. The main metric discussed in the literature (Chidamber and Kemerer, 1994; Basili, et al, 1996) is *NOA and NOM* that count a class's attributes and methods respectively. As the researchers specify high NOA and NOM denotes a class with limited extensibility and modifiability which is likely to be more application specific. Another important aspect of a method's that can be measured is the method's Cyclomatic Complexity proposed by McCabe. These are two important factors that should be measured, class size specified by its objects and methods complexity.

COHESION

A highly cohesive class's parts are highly correlated and the class itself performs only one function. Cohesion metrics can measure the class's level of cohesion by examining the relation of the class's methods with each other. High cohesion demonstrates that the class has been subdivided at an appropriate level. Low cohesion is usually known to be associated with problems and maintainability.

For measuring cohesion, the main metrics for doing so are LCOM, some variations of this, TCC and LCC. A high LCOM value signifies that the class's cohesion is low. TCC and LCC metrics signify tight and loose cohesion respectively. LCOM has received its deal of critique because as defined it measures methods dissimilarity through the instantiated variables (Gupta, 1997). In an attempt to resolve these issues LCOM2 & 3 derived that tackled these issues. As result it is decided that instead of employing LCOM, instead employ LCOM2 which as a result would provide more accurate results.

3.3.3 SELECTED METRICS

The final selection of metrics is depicted in Appendix D. The selection was based on related work that has been made on this field (see §3.2), the assessment of metric suites (see §3.1) and mainly through the classification performed in this chapter (see §3.3) in identifying particular metrics appropriate to assess each of the levels of a program. Moreover for most metrics there is none or even inadequate description in the literature to specify how they can be measured, therefore before choosing the metrics it was necessary to identify that sufficient description and details of how the metrics can be measured is available. The Table below shows the metrics chosen mapped to the concepts and mechanisms of Object-Oriented Programming.

CONCEPT	MEASURE
COUPLING	RFC, CBO
MESSAGE HIDING	AHF-MHF
INHERITANCE	DIT

COHESION	LCOM3
COMPLEXITY	CCN
METHODS	NOM
ATTRIBUTES	NOA

TABLE 1 – OO CONCEPTS MAPPING TO METRICS

Supporting the selection of metrics made, a relevant research performed on finding metric appropriate enough for identifying and measuring a project's quality by Amandeep et al. (2009), identified that some of the metrics choosen and particulary, NOA, NOM, MIF and AHF and metrics for measuring the class's methods and attributes size. Rosenberg (1998) as well colcusions regarding a relevant evaluation performed concluded that the combination of both traditional and object oriented metrics is most effective.

HIGHLIGHTS AND CONCLUSIONS

The final selection of metrics was presented after following the empirical evaluations of the metrics from the literature and based on the classification constructed. The metrics selected have been deemed as the most appropriate and have been successfully proved to provide results both accurate and reliable. The next chapter follows provides and complete specification of the metrics measurements and how the results of the metrics will be combined to generate an overall mark for the project.

ANALYSIS OF METRICS AND MEASUREMENTS

This chapter will analyse the metrics chosen to be applied in the tools development. For each of the metrics complete specification of how the measurements will be achieved and particularly how the measured values will be calculated. These measured values have not meaning unless normalised. The normalisation is the process at which the values will be transformed in the desirable range [0,100] for calculating the final mark. Different techniques are available for normalisation therefore for each of the metrics the ones most appropriate have been selected. The final section describes the calculation of the final mark.

4.1 METRICS SPECIFICATION

This section specifies how the measurements of the particular metrics will be achieved and in particular the algorithms for measuring the metrics based on the specifications of the metrics provided in the literature. Examples of how the measurement can be achieved are demonstrated with the use of a class diagram listed in Appendix L.

An apparent problem that has to be noted as identified in the literature and many researchers support is the metrics definitions. Taking as an example Chidamber and Kemmerer's Weighted Methods per Class (WMC) metric, logically it would be defined as the average of the number of methods in each class. Instead looking carefully at the definition this metric measures the sum of the methods complexity for each class. The problem that arises here is that the end results might not be as expected as the specifications for the metrics selected have been misleading.

4.1.1 SIZE

NUMBER OF ATTRIBUTES (NOA) & NUMBER OF METHODS (NOM)

To avoid the perplexity involved with the metrics specification as discussed above instead of WMC, NOA and NOM will be implemented because what is mainly required to be achieved is measuring the class size as characterised by the number of attributes and methods of the class. These metrics are straightforward and therefore do not required further specification.

4.1.2 COMPLEXITY

CYCLOMATIC COMPLEXITY NUMBER (CCN)

Cyclomatic Complexity was first introduced by McCabe. Cyclomatic Complexity is measured using the following formula:

$$CCN = decisions + 1$$

The number of *decisions* specifically is the number of certain statements used in a method and the number increases as the number of decisions in a method increase as follows:

statements	**CCN outcome**	**clarification**
if /else if	+1	if or elseif statements count as single decision.

else	0	else does not cause a new decision.
switch..case^{n}	+1 per n	each of the cases in a switch counts as a decision
switch..default	0	the default similar to the else statement does not cause a new decision
for(each)	+1	The loop is based on a decision
while/do..while	+1	Similarly to a for loop the content is executed continually if the statement is true thus a decision is made
try..catch	+1	again catch an exception is based on a condition on the type of exception to be caught if thrown

TABLE 2 – CCN DECISIONS

TCC (Total Cyclomatic Complexity) has to be calculated to retrieve the value for CCN for the class as a whole. To achieve this it can be calculated by summing the CCN's of all methods in a class and extracting the number of methods in the class.

$$TCC = SUM(CCN) - TM + 1$$

4.1.3 COHESION

LACK OF COHESION IN METHODS (LCOM)

This metric was introduced in the Chidamber & Kemerer metrics suite (§2.4.1) and then modified to LCOM2 and finally to LCOM3 by Henderson-sellers (1996). The LCOM indicates a class's level of cohesion and is measured as follows:

$$LCOM3 = \frac{M(C_i) - \frac{\sum_{j=1}^{TA} M(A_j)}{A(C_i)}}{M(C_i) - 1}$$

$M(C_i)$: Methods in class C_i
$A(C_i)$: Attributes in class C_i
$M(A_j)$: number of methods accessing attribute A_j
TA: Class Ci total attributes

The LCOM3 value varies between 0 and 2. LCOM3 greater than1indicates lack of cohesion. Therefore estimating that for `Librarian` the number of the class's attributes being accessed by the methods is 9 LCOM3 is estimated as follows:

$$LCOM3 = (11 - 9/3) / (11 - 1) = 8/10 = 0.8$$

4.1.4 INHERITANCE

DEPTH OF INHERITANCE TREE (DIT)

DIT = maximum inheritance path from the class to the root class

Taking as an example Appendix L the *DIT* for class `FacultyMember`, `Student`, `ReferenceBook` and `IssuableBook` is *1*. For the rest of the classes *DIT* is *0*.

4.1.5 COUPLING

COUPLING BETWEEN OBJECTS (CBO)

CBO = number of classes to which a class is coupled

This measures the number of other classes in the system of which a class is coupled with, the efferent coupling of the class. CBO is defined for classes and interfaces, constructors and methods. It counts the number of reference types that are used in: field declarations, formal parameters and return types, throws declarations, local variables and finally the types from which field and method selections are made.

Considering the example, and assuming the Catalogue Class contains an array of Books representing the books contained in the Library. Therefore the CBO of the Catalogue Class when measured is 1 because it is coupled to the Book Class. The `Librarian` uses the `Catalogue` and the `Member` therefore the CBO would be 2 as it updates both classes.

RESPONCE FOR CLASS (RFC)

RFC Response set of class is the total number of methods that can be invoked when a message is sent to an object of the class. This includes all methods of the class and any methods executed outside of the class as a result of this message. More simply the response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class, meaning the number of methods that will be invoked on a message call.

4.1.6 INFORMATION HIDING

AHF and MHF metrics are both used for measuring the information hiding concept of object-oriented paradigm. Specifically the formulas have been listed exactly as described by Ja'Afer and Sabri (2005) in the literature.

ATTRIBUTE HIDING FACTOR (AHF)

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

$A_h(C_i)$: hidden Attributes in class C_i
$A_d(C_i)$: $A_v(C_i)+ A_h(C_i)$: Attributes defined in C_i
$A_v(C_i)$: visible Attributes in class Ci
TC: Total number of Classes

AHF as specified is calculated by applying the formula above. In particularly the formula measures the number of hidden attributes and is divided by the total number of the class's attributes. More specifically a class that has all its attributes accessibility as Public results in AHF = 100 in a range [0,100]. The higher AHF is the lower the resulting mark will be.

METHOD HIDING FACTOR (MHF)

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

$M_h(C_i)$: hidden Methods in class C_i
$M_d(C_i)$: $M_v(C_i)+ M_h(C_i)$: Methods defined in C_i
$M_v(C_i)$: visible Methods in class Ci
TC: Total number of Classes

Similarly to AHF, MHF as specified is calculated by applying the formula above. In particularly the formula measures the number of hidden methods and is divided by the total number of the class's total methods. Particularly, in a case at which a class declares all its methods as private

MHF= 0; in the [0,100] range. This concludes that the lower MHF is the higher the resulting mark will be for the class accordingly.

Finally, it is important to list here an important issue when measuring systems performance identified by Benlarbi et al. (1999). The issue identified by these authors it the impact that the class's size has on the results. From their analysis using the CK metric suite, they reached to the conclusion that size can have an important and confounding effect on the validity of object-oriented metrics. It is important therefore when the testing of the system is carried out to verify whether taking into account the classes' size would improve the results accordingly, since this could significant impact on the results otherwise.

4.2 METRICS NORMALISATION

Normalising measured values is a vital part of computing the final mark, since the measured values, as is, have misleading or unclear meaning. Calculation of an overall mark for the code quality therefore requires initially, the normalisation of the measured values in a range between [0, 1]. After having a look at the metrics we reached to the conclusion that there should be two possible ways for normalising the values as demonstrated in the graphs below:

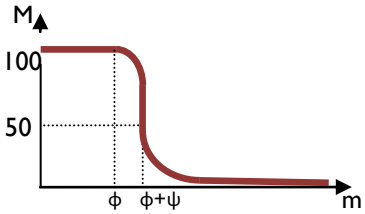


CHART 1 - FIRST APPROACH

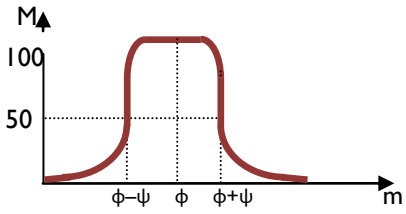


CHART 2 - SECOND APPROACH

First of all, metrics such as NOA, NOM and CCN, the lower their measured values are the better the results are. The higher the value means the lowest the code quality is. For normalising the value consequently in the range of [0, 100] as the measured value increased within the Upper and Lower Boundary values m[min, max] then the lower the normalised value should be in the M[0, 100] range.

Finally, other metrics, and particularly size metrics should be normalised as displayed in Chart 2. Taking as an example a class's size should be too small or too big. Taking this into consideration an average size class is most preferable. By employing this normalisation way the closer the size of the class - as measured (depicted as m) - is, to the average of the min and max boundary values [min, max], the higher the resulting mark will be (depicted as M).

After normalising the values for each of the metrics the weighted and final marks are required to be calculated. The next chapter introduces how the final mark will be computed by applying the weights to the measured values after being normalised.

4.3 WEIGHTED AND FINAL MARK COMPUTATION

After normalising the values, for calculating the final mark it is necessary first to calculate the weighted marks which will apply the weights to the measured metric values. The range for the weighted values will be from 0 to 1. The WEIGHTED MARK (WM) will be calculated using the formula:

$$WEIGHTEDMARK = MV * MW$$

MEASUREDVALUE (MV): The calculated and normalised metric value
METRICWEIGHT (MW): The specified weight for the metric [0, 1]

Thereafter, the final mark will be calculated by summing all the weighted marks of the different metrics and dividing the sum by the total number of metrics used in the evaluation of the code as specified in the formula below:

$$FINAL\ MARK = \frac{\sum_{i=1}^{n} WEIGHTEDMARK}{NM}$$

WEIGHTED MARK (WM): The weighted mark calculated from the formula depicted above.
NUMBER OF METRICS (NM): The number of metrics been selected for the evaluation

HIGHLIGHTS AND CONCLUSIONS

This chapter gave a complete description and specification of the metrics selected through examples. The normalisation techniques that will be used for normalising the metric's measured values have been identified. Finally two formulas have been specified, one for applying the weights on the normalised values and one for computing the final mark.

PRACTICAL WORK

This chapter will be an overall discussion of the practical work done. It includes a discussion for each phase of the software development life cycle for the product been developed, including the definition of requirements, the design documentation, the product's implementation and the testing process. The discussion outlines the choices made for the consideration of any tools and technologies used (languages, IDEs, third-party components used, etc), along with any alternatives considered and the reasons for those choices.

REQUIREMENTS SPECIFICATION

This chapter focuses on the requirements specification. Firstly the necessary properties are listed to provide a complete overview of the metrics selected that will be necessary when developing the metrics.

5.1 METRICS SPECIFICATION

5.1.1 METRIC PROPERTIES

Each of the metrics has particular properties. These properties as identified in the literature review include:

- Lower Boundary – The minimum value of that the measured value of the metric can have
- Upper Boundary – The maximum value of that the measured value of the metric can have
- Weight – The impact that the metric should have on the final mark.
- Measured Value – The calculated value when measuring the specific metric
- Name – The specified name of the metric
- Acronym – Metric name abreviation
- Description – A description of the metric necessary for displaying information about the metric
- Normalisation Code – The code that signifies which formula to be used for normalising the measured value (see §4.2)
- System Level – This signifies if the metric analyses classes at the system level
- Class Level – This signifies if the metric analyses classes at the class level

5.1.2 BOUNDARY VALUES

It is necessary to specify the boundary values both lower and upper for normalising the metric values as discussed in §4.2. These values have been identified through previous empirical investigations.

Limit	Explanation
DIT <= 5	Inheritance tree should not exceed a reasonable number
AHF <= 3	The class variables should be declared as private.
MHF< =3	The class methods should be declared as private
200<LOC>1	The class file should not be empty but not exceed 200 lines of code
NOA [1,8]	There should be at lest 1 method
NOM [1,12]	There should be at least 1 method in each class but the number should not exceed 12
LCOM3 <= 0	The higher LCOM it indicates that the class has low cohesion
CCN <= 10	The Cyclomatic Complexity Number of any method should not exceed 10
CBO <= 5	Class coupling should be limited to 5 otherwise the class will be difficult to extend or maintain.

TABLE 2 – METRICS BOUNDARY VALUES

5.1.3 WEIGHTS

The weight is an important factor for calculating the final mark and getting the required results. By specifiying weights to each of the metrics, it can be specified the percentage that each of the metrics should count towards the final mark. The metric values will range from 0 to 1 where 0 means that the metric should not count towards the final mark.

5.2 TOOL REQUIREMENTS SPECIFICATION

There are a lot of factors to consider before the initiation of the tools development. Such factors include usability, configurability, flexibility and in general the functionality that the system will provide. Usability is not the main concern but for every system development a well designed interface provides a better system. The system should provide the availability of selecting necessary metrics since different assignments have different requirements and the availability of the tool to dynamically be configured when adding more metrics. The system should be able to store the different configurations of the metrics. All these requirements need to be analysed in more depth to provide a comprehensible view about how each of them can be developed to provide a functional solution. This chapter specifies the requirements of the system as identified.

5.2.1 USER-SYSTEM INTERACTIONS

Identification of the user interactions with the system can be achieved by graphically depicting this interaction. With a Use Case Diagram the system's behaviour can be described and further analysed.

USE CASE DIAGRAM

The Use Case Diagram was designed to illustrate the tasks that the user performs when using the system, the functional requirements. These requirements are depicted in the Use Case diagram in Appendix I. For each of the requirements a brief textual description is given in the table below giving specific details of how each of the functions is achieved the purpose and its goal(s).

Use Case	Description
load java class/project	The user can either load a single java class or a folder as a project to evaluate. The java class(es) are loaded and displayed on the screen.
change properties	As the metrics have different properties the user can change these for the tool to use when evaluating the code.
save properties	The properties set by the user can be stored with specific names allowing this way the user to specify particular properties for different assignments
load properties	These properties can be loaded at any time to be used for the evaluation
select metric(s)	The user has the choice to decide on which metrics to use and which to exclude
run evaluation	The user has to click the run button for the tool to calculate the metrics
view results	The results of the evaluation are displayed on the screen for the user to view
load metric class	The tool has been designed to be flexible. Thus the user can write up a class that will represent a metric and load the class into the tool. The tool automatically finds the class and will be usable at any time

TABLE 3 – USE CASES AND DESCRIPTIONS

The system developed has been designed to offer the availability to the user to select which classes or projects to load for evaluation amend the properties of the metrics and save them in specific files for re-using when needed.

Mainly it is important to state how some of the requirements have been established. Allowing configuration of the metric properties was based on the fact that no established framework exists when marking student assignments in the literature and therefore the tutors can choose upon their own personal preferred frameworks the metrics appropriate. Also by allowing configuring weights upon the metrics the tutors could put emphasis on specific aspects of object-oriented programming. Finally, by allowing the metric boundaries configuration will allow the tutors to raise or lower the boundaries for specific assignments when the overall mark is too high or two low depending on the assignment. Another important requirement was the dynamic loading and invocation of the metrics to provide extensibility of the tool to a certain level.

Resulting from the Use Cases, a comprehensive list has been constructed denoting the required functionality in different levels as exhibited in the next chapter.

5.2.2 REQUIREMENTS

From the analysis for the system six (6) basic development requirements have been identified as the basic requirements for assessing the code quality based on object-oriented metrics:

- The system must be able to **load a single java file or multiple java classes** as a project.
- The system must be able to **dynamically load metric classes** that adhere the per-defined properties
- Each of the **metric classes must implement their specification** (§4.1) and therefore to be able to evaluate the class.
- The system must be able to **store and load the metric properties** to be used during the analysis
- The system must be able to **analyse the class(es)** and **generate a measured value** for each of the classes for each of the specified dynamically loaded metrics.
- The system must be able to **normalise and create the final mark** by combining and calculating the measured values that will reflect the overall quality of the code

This list of requirements will be employed to generate a structural development framework for developing, designing, implementing and testing the system. The following chapters continue on analysing the requirements specified to attain a more complete overall view of how exactly we will progress with the system's analysis and development.

5.3 DEVELOPMENT BASIS

It is necessary first to set the grounds for the development before initiating with the design and implementation of the system.

5.3.1 METHODOLOGY

To succeed in the development of the system a methodology has to be specified to provide a framework that will be used for the arrangement, planning, and control over the process of developing the system. The methodology that has been decided to be used for the development is Agile (Extreme Programming) methodology which is based in Iterative development as shown in Figure 3.

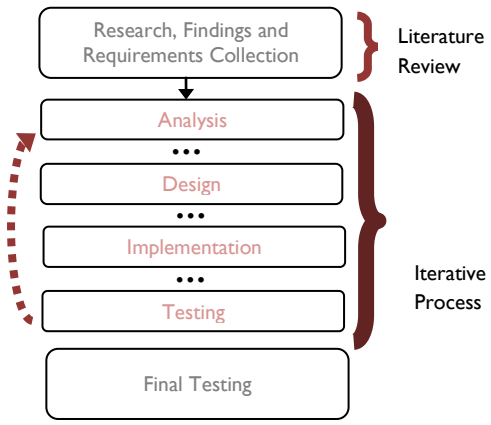


FIGURE 3: DEVELOPMENT METHODOLOGY ITERATIVE DEVELOPMENT

Iterative Development normally is the type of development that cycles through the phases, from requirements gathering phase, to implementing functionality in a working final release. Instead for this project since the requirements have all been gathered initially it was slightly ammended and the Iteration guide through only the practical's work phases; *Analysis*, *Design*, *Implementation* and *Testing*, and at the end a final overall testng will be performed.

Using this methodology the system will be decomposed into functional parts and implement the in single parts for which one there will be an iteration through the development cycle and finally the individual parts will be combined to a single and form the system. Supporting this decision was that the Iterative Development processes grew out of Object Oriented Development, and using Java's Classes, Objects the requirements can developed in isolation, and each of the task will be naturally boxed by its responsibilities.

5.3.2 SYSTEM ARCHITECTURE

A lot of frameworks exist upon which a lot of applications are built. These frameworks specify different architectures that by being utilised can improve significantly the design of a system and therefore are usually applied. For the design and implementation of the system many different are available but there is one specific that would be the most appropriate, the MVC architecture also entitled as design pattern.

The MVC (Model-Vie-Controller) architecture was visualised and created by Trygve Reenskaug (Reenskaug, 1989). This design pattern was originated in desktop applications, and relatively recently made its way into web based applications. It was originally built to manage the Interface and interaction with the user.

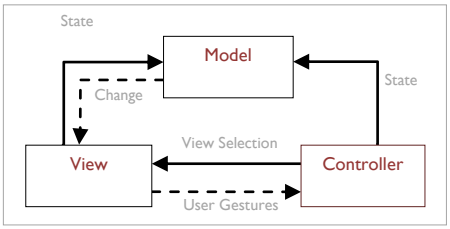


FIGURE 4: MODEL-VIEW-CONTROL

Model-View-Controller minimises the coupling between the business logic and the data model by dividing the applications into three layers as depicted in Figure 4:
Model: maintains and manipulates the data (Turner and Bedell, 2003). It provides updated information to the view layer and also responses to any queries from the control.
View: represents the interface. It renders and updated the User Interface whether it is a graphical user interface or a command editor.
Controller: Controller mainly manages the updates to the views (Bennett et al., 2006). It maintains conditional logic on which information will be displayed to the user, querying the model and signalling the view to be updated.

MVC is mainly used for interactive systems (Bennett et al., 2006). Mainly MVC and design patterns in general, are used because it has become widely accepted among software engineers and architects that designing applications that conform to these patterns facilitates the re-usability, maintainability by achieving low coupling and high cohesion. Furthermore, MVC, on the positive side, provides re-usability of model components, ease of support for new types of clients with the price of increased design complexity (Java, 2002).

The design problem that MVC solves is that of simplifying three primary functions that are common in many applications (Turner and Bedell, 2003), and thus the three tiers. Mainly at the business level the application of MVC disseminates the work and separates the interface from

the core processing thus the work can be divided between the working groups. Bennett et al. (2006) highlights that in contrast with other architectures such as the four layer one, MVC ensures that all view components are kept up to date, by separating the core functionality (model) from the interface and through its incorporation of a mechanism for propagating updates to other views.

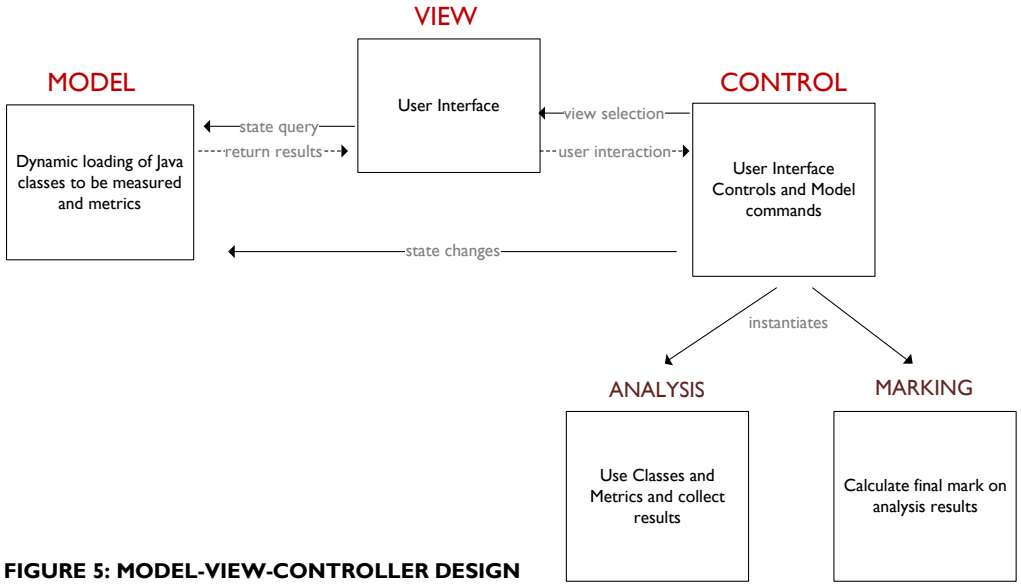


FIGURE 5: MODEL-VIEW-CONTROLLER DESIGN

This model above (Fig. 5) has been constructed as the initial step in understanding the concept of the system being developed and demonstrates the interaction between the layer and further decomposition of the model layer.

5.3.3 DEVELOPMENT TASKS PRIORITISATION

Due to that their must be a control over the development of the tool thus their must be a coordination and control over the development of the tool's different features. This could be specified using MoSCoW rules which is a prioritisation methodology for classifiying and ordering the different features according to which must be implemented, which should be considered if possulbe, which could be considered and won't be implemented since there are additional features of the tool. MoSCoW rules, despite that is a very simple methodology of prioritasing tasks and features, can been proven to be very effective since despite the ordering there is control over the development because through this method acommon understanding can be reaced on the importance that should be placed when delivering the requirements.

MoSCoW capitilised letters define the four classifications of this methodology:

- MUST HAVE

  Implementation of enough metrics to cover all object-oriented aspects

  Ability to load and save metric configurations

  Normalisation of metric values

  Ability to choose both indivual java files and projects

  Metrics for measuring cohecsion, coupling, inheritance, complexity.

SHOULD HAVE

Metrics for measuring polymorphishm, information hidding, considering the complication in their implementation

Graphical User Interface for the enhancement of the tool's usability

COULD HAVE

Flexibility on the tool to support easy addition of new metrics

WON'T HAVE

Some validations can be made as assumptions. For example using Java compliler the classes can be compiled and checked if they actually can be run. Later in this chapter these assumptions will be listed

5.3.4 DEVELOPMENT PREQUISITES

The discussion outlines the choices made for the consideration of any tools and technologies that will be used for the development (languages, IDEs, third-party components). The programming language for developing the system is Java. Java is widely acknowledged and nowadays it has been one of the most commonly used languages in the industry and for teaching programming. Since the system will be used for marking Java assignments making use of the Java for the implementation of the system is perceivingly the most appropriate choice. The tool for the development is BlueJ a simple yet powerful environment which provides an interactive environment initially being developed for teaching Java to novice programmers.

Further since it was decided to proceed on implementing the system to provide a Graphical UI for the Tutors to interact with a 3rd party libraries have been identified for improving the representation of a Java class code when displayed. This is the `java2html` library (www.java2html.com) that converts Java Source Code into a colourised and browsable HTML representation.

Last as already discussed the code analysis will be performed statically by employing certain libraries that are available. The selection will be made later on, in the design phase to select which of the libraries is more appropriate and better for the project.

5.4 ASSUMPTIONS

For the design and implementation certain assumptions have been made. Especially for when the metrics where implemented. Such include:

- The classes loaded should be already be compiled successfully.
- The classes should not be empty
- The class files generated from the compilation must be contained in the same folder as the java file.

HIGHLIGHTS AND CONCLUSIONS

This chapter provided the list of requirements that need to be designed and implemented in the tool being developed. The necessary metric properties have been listed and the overall systems requirements and the basis for the development have been set up. The iterative development methodology has been chosen to be the most appropriate one and the MVC framework as most suitable to use as the foundation to build the system upon. As perquisites the Java will be the programming language and the system will be developed with the BlueJ IDE. Certain assumption have been made clear to specify certain condition that need to be satisfied for the system to work.

SYSTEM DESIGN

6.1 SYSTEM DESIGN

This chapter demonstrates the system's design. The design process it one of the most important parts when developing any system as good designs usually derive to well-developed system. The importance lies in the fact that through this phase critical decisions have been taken on which approaches to employ that could affect the development positively or negatively depending whether the right decisions have been taken. The chapter investigates different approaches for storing/accessing the system's properties and approaches for implementing the functionality of the metrics for analysing the code statically.

6.1.1 CONCEPTUAL SYSTEM DESIGN

The tool has been designed to support object-oriented software quality assessment and marking. The diagram below was developed based on the requirements defined §5.2.2 mainly to depict the concept. It demonstrates as the concept the idea based on the requirement. More in particular, using the source code loaded each of the classes is analysed by the metrics from which the measured values as derived. These results are used for calculating the total mark and then it's been displayed.

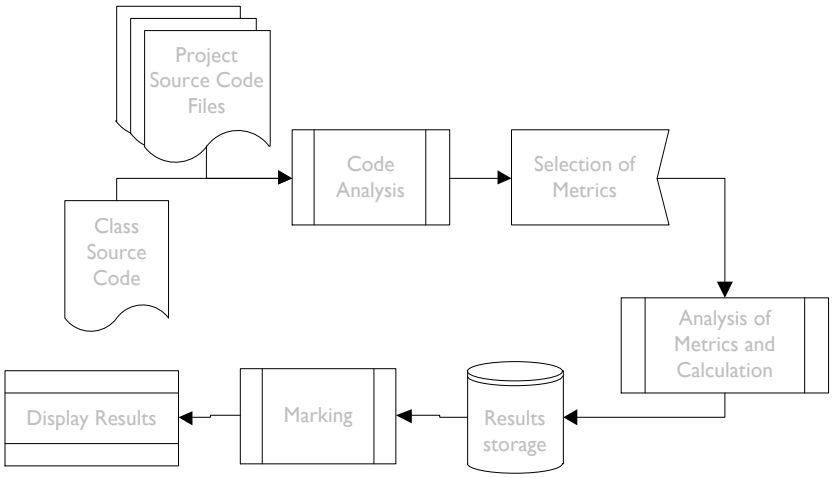


FIGURE 6: SYSTEM DESIGN

6.2 INVESTIGATION OF POSSIBLE APPROACHES

One of the most important parts of the development is the investigation and identification of approaches for using and implementing the system's requirements. The importance of this investigation lies in that without having a clear look at the possible approaches wrong decisions might be made which would affect negatively the project. The main parts that will be investigated is how

the configuration of the metrics will be stored and how the static code analysis of the Java classes can be achieved. These are the points discussed in the following sub sections.

6.2.1 CONFIGURATION AND STORAGE

For the tool been developed to provide the necessary flexibility and ease the use of the user with the system, by providing the availability for storing the current configurations of the system. These configurations include the metrics for the weights, the metrics boundary values, which metrics should be used and moreover it could also be used for specifying the comments for providing feedback on the results of the evaluation. There have been three approaches identified for managing, storing and accessing the required configurations, that include ***Java Architecture for XML Binding***, ***Java properties (.properties) file***, and finally using the functionality provided by ***Java for Reading/Writing text (.txt) files***.

Java Architecture for XML Binding (**JAXB**) (Ort and Mehta, 2003) allows the mapping of Java classes to XML representations. JAXB can be used with two different ways, to cast Java object as XML and inversely from XML to a Java object. JAXB does not require having specific load and save methods for structuring the class by retrieving the data from the memory. With XML Binding an object can be developed in Java that will hold the configurations specified and even if the properties change the XML file will not require any amendments.

Java's Properties are values that are handled in key-value pairs which can be stored in a .properties file. Both the key and value are represented as Strings. Taking as an example the system being developed, one of the metrics CBO properties is named weight therefore the key assigned will be "CBO.weight" and the value the required value ("CBO.weight = 20"). For managing the properties, Java provides the `java.util.Properties` class that includes functions for retrieving and listing the values and saving them again to any filename given with a '.properties' extension.

Another functionality provided by Java is the `FileReader` and `FileWritter` classes that allow the ***Reading/Writing of text (.txt) files***. Using a specified arrangement for writing to a text file the configuration properties can be stored and then could be retrieved when needed by reading the file using the Scanner class provided by Java. There is no control over reading and writing and many things could go wrong using these classes and would even be difficult to extend and maintain if more properties were to be stored in the file.

There is an apparent increase in the utilisation of XML in the modern world because of the capabilities offered by XML that can be used for storing complex and potentially incomplete data sets. However, using XML to store simple configuration information is excess and makes the configuration file needlessly complex to edit and needlessly complicates the application. Unless there is need for something complex the configuration file does not need to be complicated. Considering also using a text file would be more time consuming and risky than using the .properties file functionality already provided by Java that already provides the

Properties class to handle the configurations as objects. All that is required is the storage of name-value pairs to specify a few parameters for which the most helpful and simple solution is the .properties file that provides both the necessary and required functionality and simplifies the development.

6.2.2 STATIC CODE ANALYSIS

Three possible approached have been identified for analysing and evaluating the code. These are the using the ASM Library, the Byte Code Engineering Library (BCEL) and finally Java Reflection. ASM and BCEL are both libraries developed for providing functionality similar to the one provided by Java's Reflection but they are specifically developed for working with bytecode and classfiles. Reflection provides a way of being able to access Objects and Classes dynamically and therefore use information that is located externally and can be used at run time. The drawback though, is that reflection is relatively slow and even compared to ASM and BCEL.

Class-working techniques provided by ACM and BCEL offer an alternative. Instead of using reflection this can be achieved accessing the class file of the object and can be performed relatively much faster. From a first look at both libraries both offer similar functionality but ASM appears to be relatively faster. Despite that, after a first attempt to use ACM it has been proven to be very complicate to use because of the lack of good documentation and examples. On the other hand BCEL provides comparatively a better documentation and examples for using as a starting point for understanding how it can be used. Therefore BCEL has been chosen to be used for analysing the classes and calculating the metrics.

USING BCEL

Understanding how to use BCEL requires first to look at how a java file is transformed into a class file which file BCEL uses to provide this functionality. Java Programs are compiled into a portable binary format called *byte code*. Every class is represented by a single class file containing class related data and byte code instructions. The layout and format of a Class file as generated is depicted in Figure 7 below.

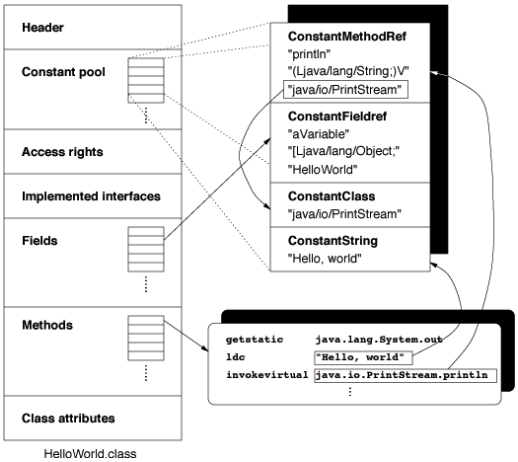


FIGURE 7: JAVA CLASS FILE FORMAT

A class file is composed of the components as named on the Figure 7. The constant pool basically holds the following types of constants: References to methods, fields and classes, strings, integers, etc. Each class methods define a set of byte code instruction which consists of 212 instructions and 44 opcodes. These instructions can be grouped in Stack Operations, Arithmetic Operation, Control Flow, Load and Store Operations, Field Access, Method Invocation, Object Allocation and Conversion and type checking. Further information about the bytecode was available in Lindholm and Yellin (no date) online book that provides a full specification of the Java Virtual Machine, and more specifically the instructions and instruction sets and the loading, linking and initialisation of classes and interfaces which will be necessary for when developing the metrics.

6.3 UML ANALYSIS AND DESIGN

Unified Modelling Language (*UML*) is a standardised general-purpose modelling language in the field of software engineering. Modelling as an activity has been utilised over many years for system's development. By defining a model makes it easier to break up and decompose the system and divide its complexity into discrete pieces that can be studied individually. We can focus more easily on the smaller parts of a system and then understand the overall picture. This is the concept upon which the development was based upon as well, by developing smaller parts and then combining everything at the end.

6.3.1 DATA FLOW DIAGRAM

The data flow diagram (DFD) has been designed to provide a graphical representation of the "flow" of data through the system (see Appendix I). The diagram has been separated into five (5) Workflow sections based on the requirements defined previously in the specification of the system's requirements (see §5.2).

The main advantage of using the data flow diagram is that it helped in understanding better the processes of the system as individuals by highlighting and analysing the requirements and more generally provided an overall description of how the system's functionality was achieved.

6.3.2 CLASS DIAGRAM

Moreover a more detailed high-level description of the system was required to describe the how the features of the system will be implemented in an Object-Oriented manner and achieve high decomposition of the system where at the same time providing the required functionality. The system designed using the MVC pattern as noticeable in Appendix J and therefore the system is broken down into three (3) packages, one for each layer. The view constructs the User Interface, the Model executed all the queries for loading, parsing the java class files and analysing, normalising and measuring the classes. The control handles the listeners for the User Interface's control and queries accordingly the model. Also, as shown a

driver class initialised the three layers. Finally the class diagram displays three more packages, the *libs*, *metrics* and *properties*. The libs package contains all the 3rd party Libraries that have been used and required for providing certain functionality, the metrics contains all the implemented metrics and the properties the saved properties files.

6.4 DEVELOPMENT PHASES DESIGN

The fundamental requirements have already been defined in §5.2.2. Cycling through each of the requirements a series of phases has been defined where at each phase each of the requirements will be analysed, to gain an understanding of what will be done in each phase and their outcome.

6.4.1 PHASE 1

> *"The system must be able to load a single java file or multiple java classes as a project."*

For this requirement there are two different parts to be developed; loading individual files and loading a project. Loading a file is different than loading a folder as when loading a folder all the files in the folder and its subfolders must be iterated through and identified. Using FileChooser provided by Java this can be achieved for both cases since it can be set to read either files or folders. When choosing a folder a recursive method can be constructed that while go through all of the folders and subfolder and find all the .java files.

6.4.2 PHASE 2

> *"The system must be able to dynamically load metric classes display them for the user to modify their properties"*

As defined in §5.2 the system must accommodate flexibility by allowing to add new metric classes that can be automatically loaded into the system and used in the evaluation. Therefore the metrics have to be implemented as *stand-alone* classes. By defining a class as stand-alone its meant that the class itself will not rely on any other classes for providing the required functionality. For managing the dynamically metric execution it has been decided to implement an Interface that would enclose all the functions and properties required for each of the metrics to implement. Therefore by implementing an interface it will allow the metric classes to become more formal about the behaviour they are required to provide.

6.4.3 PHASE 3

> *"Each of the metric classes must implement their specification (§4.1) and therefore be able to evaluate a class and produce the measured value."*

The metrics must implement certain logic for evaluating the code and use the formulas identified (§4.1) for measuring each of the metrics to accomplish retrieving reliable information and data. For achieving the measurement of classes using the metrics BCEL (see §6.2.2) will be used. As each of the metrics has different specification the BCEL will be used accordingly to provide those measurements. Each of the metrics will be developed individually in isolation as different sub-phase under this main phase to provide sufficient details of how each of them has been implemented.

6.4.4 PHASE 4

> *"The system must be able to store and load the metric properties to be used during the analysis"*

This requirement has to been broken down into three (3) sub-phases, (a) the development of the saving part, (b) the development of the loading part and (c) applying the properties to the metrics. From the analysis in §5.1.1 all the required metric properties have been defined and therefore using the approach identified and selected in §6.2.1 of how to store and load the properties this requirement can be implemented. Applying the metrics properties can be achieved dynamically using reflection and therefore the metrics can use of the properties being set and as defined by the Interface discussed above (§6.4.2).

6.4.5 PHASE 5

> *"The system must be able to analyse the class(es) and generate a measured value for each of the classes for each of the specified dynamically loaded metrics."*

As defined by this requirement the properties loaded can be used by the individual metrics for when evaluating the code by applying the metrics as discussed in Phase 3 (see §6.4.3). The system by invoking the methods specified in the metric interface. Therefore the results can be collected that will be used in the following phase for normalising and calculating the final mark.

6.4.6 PHASE 6

> *"The system must be able to normalise and create the final mark by combining and calculating the measured values that will reflect the overall quality of the code"*

The normalisation of the metric values has been discussed in §4.2. The normalised values will therefore on be used to calculate the final mark as discussed in §4.3. A class will be constructed for achieving this, separating this way the functionality between objects.

6.4.7 FINALISATION PHASE

The final phase is meant for employing the MVC pattern as the system's architecture. Using MVC offers a lot of benefits as discussed in §5.3.2. More specifically it will allow the easy expansion and alteration of the system by dividing the system into the three layers. Therefore the system architecture will be divided more specifically into the three layers of which purpose will be:

MODEL: Will represent the model of the system. This package will contain the classes responsible for loading the class files and the metrics, using the metrics evaluate the classes, normalise the measured values, calculate the return the results.

VIEW: The view represents and constructs the User Interface (UI). By using the MVC framework the view provides that flexibility that it can be modified from a command based UI to a Graphical UI and contrariwise.

CONTROL: The control works as the intermediary between the model and the view, as it receives the requests from the view queries the model and returns the results if any for updating the view.

HIGHLIGHTS AND CONCLUSIONS

Through this chapter important decisions have been made and the system was design using the UML approach. The decisions made included the approaches that will be used for storing the metric properties and more importantly the approach selected for analysing the code. BCEL has been found to be simpler than ACM in the comparison despite the difference in performance and based on the time constraints it was considered as more appropriate. Moreover during this phase a Class and a Dataflow a model that made easier breaking up and decomposing the system by divide its complexity into discrete pieces that can be studied individually. Since the iterative methodology has been selected, the design for each of the phases has been discussed of what to be expected from each phase. The next chapter describes how the system was implemented in detail in regard to the phases specified.

SYSTEM IMPLEMENTATION

This chapter describes the overall implementation of the system and specified for each of the development phases the wrong done in detail and important issues tackled through the process.

7.1 IMPLEMENTING THE REQUIREMENTS

Relying on framework and the phases specified in §6.4 similarly the implementation phases followed. The following sections describe the work done and how the required functionality has been established.

7.1.1 PHASE 1

For achieving loading all the classes the `FileChooser` control that is available. The main complicate point in achieving loading the file is when choosing to load a folder and finding all the classes within the folder. This could easily be achieved using a **recursive** method that would loop though all of the items and if the file extension is .class store it in an array, otherwise if the item is folder the method would recursively invoke itself to collect all the files in that folder. This recursive iteration reaches its end when all the .class files in the folder have been collected. When a file is found with a .class extension it is stored in an `ArrayList` of type `<Class>`. The reason for this is that using Java's Collection and particularly ArrayLists, have no restrictions in size in contrast with an Array that its size has to be specified on initialisation. Moreover in contrast with Arrays, adding an item to an `ArrayList` – one of the components provided from Java.util.Collections - is much more simpler as the add method of the `ArrayList` automatically adds up the items as if it is a queue and therefore the place where it has to been added does not have to be specified.

```
01.         /**
02.          * recursive method loops through all folders of
03.          * specified filepath and finds java class files
04.          */
05.         public void findClassFiles(File dir) {
06.             if (dir.isDirectory()) {
07.                 File[] files =  dir.listFiles();
08.                 for (int i=0; i<files.length; i++) {
09.                     try  {
10.                         if (!files[i].isDirectory())
11.                             if (getExtension(files[i]).equalsIgnoreCase("class"))
12.                                 classFiles.add(files[i]);
13.                     } catch(Exception e) {}
14.                 }
15.                 String[] children = dir.list();
16.                 for (int i=0; i<children.length; i++) {
17.                     findClassFiles(new File(dir, children[i]));
18.                 }
19             }
20.         }
```

7.1.2 PHASE 2

To accommodate the flexibility required the `metric` interface has been constructed that is displayed in Appendix K. The interface specified certain methods (shown in figure 8 below) mainly get/set methods based on the metrics properties established in §5.1.1. A metric class implements the interfaces and should implement the interface's methods hence a specific set of methods is establishing for accessing the metric classes dynamically.

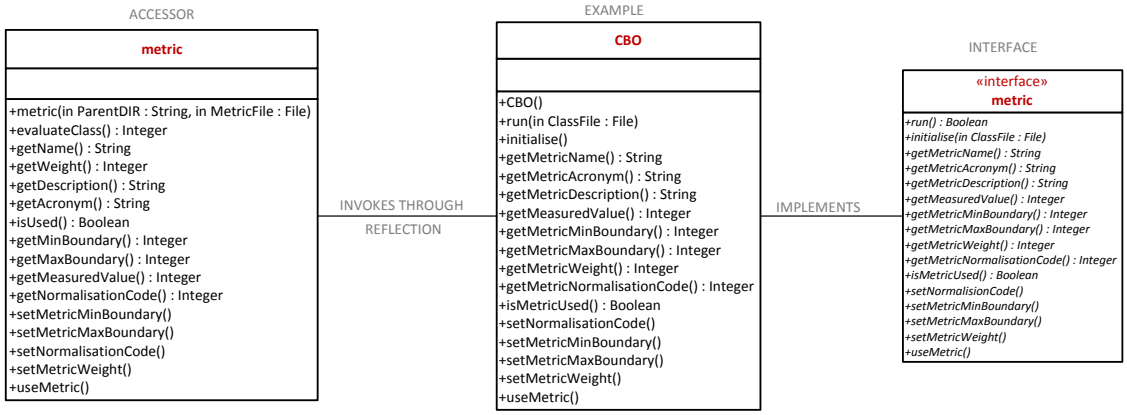


FIGURE 8: HANDLING DYNAMIC LOAD AND ACCESS TO METRIC CLASSES

Having as aim to simplify the interaction, handling and to minimize the bulk of code that has to be written each time - to execute the dynamically loaded metrics methods - the `metric` (ACCESSOR) class has been constructed (Fig. 8) which works as an *accessor* for the metrics . When the metrics are loaded for the first time for each of them is parsed into a new Metric object that acts as a container and accessor.

The metric (ACCESSOR) class when initialised takes as parameter the metric's java file (e.g. CBO.java) and the parent dir at which the metrics are located. Using this information and reflection the metric's methods and attributes can be accessed. To successfully load the metrics a `URL` must be specified pointing at the location of the directory where the metrics are located, which `URL` will be passed as a parameter to `URLClassLoader`. Then a new `Class` object is constructed which finds and loads the metric using the `URLClassLoader` specified previously using a `filepath` in the form of packageName.metricFilename (`metric.CBO`) which is basically the packageName (metrics) and the metricFilename (e.g. CBO). The `filepath` is required in this form for the metrics to be loaded successfully because the metrics are located in a package.

```
01.  URL[] fpURL = new URL[] { new URL("file:///"+parentdir) });
02.  URLClassLoader loader = new URLClassLoader(fpURL);
03.  String convertedFilepath = packageName + filename;
04.  Class c = Class.forName(convertedFilepath, true, loader);
05.  Object iClass = c.newInstance();
06.
07.
```

The metric (ACCESSOR) also contains methods for dynamically invoking the methods of any metric class. The code below demonstrates a *generic* method build for this purpose that takes as parameters the methods name and dynamically invokes it and returns the returned value. The method uses the `iClass` Class object which was defined when the object was created and was stored initially.

```
01.   private String invokeMethodReturnString(String methodName, String value)
02.   {
03.       try {
04.             java.lang.reflect.Method method = c.getMethod(methodName);
05.             value = method.invoke(iClass).toString();
06.          }catch (Exception e) { e.printStackTrace(); }
07.       }
08.   }
09.
10.
```

Similarly, the accessor class invokes the method for evaluating a class but passes as a parameter the `classfile` that will be evaluated.

```
01.   java.lang.reflect.Method init = c.getMethod("initialise", File.class );
02.   java.lang.reflect.Method run = c.getMethod("run");
03.   init.invoke(iClass, classfile);
04.   run.invoke(iClass);
05.
06.
```

7.1.3 PHASE 3

Each of the metrics has been designed to implement the `metric` interface and extend the `bcel.classfile.EmptyVisitor` class. The reason for overriding the `EmptyVisitor` class's methods is to be able to access the class's methods, attributed and instructions in *sequential visits* based on the method calls of the EmptyVisitor (Appendix N). Implementing the metrics was one of the most complicated phases of the implementation not only because of the complexity associated with using BCEL but also in successfully implementing the metric specification to achieve the measurement of each of the class's accurately and reliably. To achieve the requirement it was first necessary to understand how using BCEL could be used by examining its documentation to identify exactly how the required results could be attained. It was firstly necessary to understand how a class file is structured and how BCEL using this infrastructure can analyse the classes as identified in the static code analysis in §6.2.2.

Therefore as BCEL supports the *Visitor* design pattern, this has been employed for writing visitor objects to traverse and analyse the contents of the class files. First it was necessary to pass the Java class file of the class being analysed using the `ClassParser` object that is capable of parsing binary class files. By parsing the object a `JavaClass` object is returned that is consisted of fields, methods, symbolic references to the super class and to the implemented interfaces. Access to these fields is then achieved using the `GlassGen`, `ConstantPoolGen`

part of the BCEL API (package `org.apache.bcel.generic`) demonstated in Figure 9 below in combination with the functions provided by the `EmptyVisitor` class referred to above.

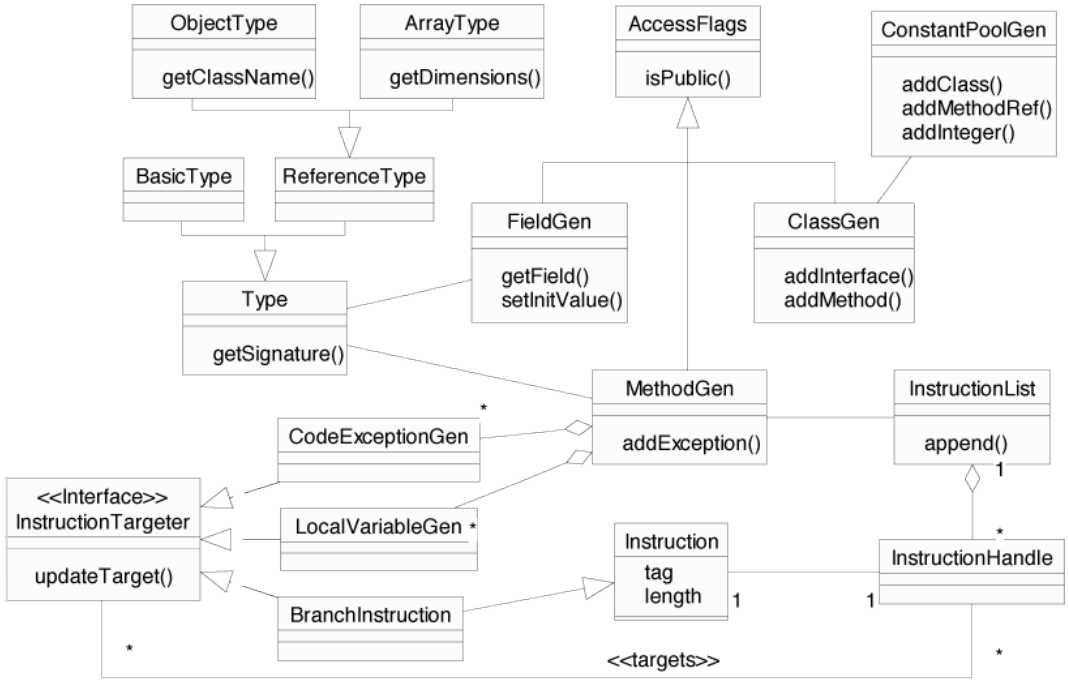


FIGURE 9: UML DIAGRAM OF THE CLASSGEN API

Summarising, each of the Classes being analysed is parsed and the `ConstantPoolGen` is constructed using the parsed class. Then by traversing through the class by implementing and sequencially invoking the `EmptyVisitor's` methods, the class's methods, fields etc. can be achieved in combination with the `ConstantPoolGen` object being constructed initially.

Each of the metrics was developed individually in a different sub-phase. Initially for designing the metric its specification as discussed in §4.1 was analysed to identify exactly what parts of a class should be measured and identify then how this could be achieved using BCEL. Whilst each of the metrics were developed at the same time testing was performed to ensure that the resulting measured value was as expected. Some of the classes were developed based on a testing approach: by monitoring the bytecode of a class helped in discovering the names of particular Instructions as defined in the VM specification for achieving the measurements.

7.1.4 PHASE 4

This requirement has been broken down into three (3) sub-phases; Saving, Loading Properties and Configuring metrics based on the loaded properties. The `Properties` class provided by Java has been used – as specified in section §6.2.1 – that represents a persistent set of properties. The `Properties` can be saved to a stream or loaded from a stream. Each key and its corresponding value in the property list is a string. This part of the system was relatively

easy to implement as the `Properties` Object contained all the required methods for storing loading and accessing the properties.

SAVING

The saving phase works in two steps, extracting the properties from the table where there are being displayed and storing the properties in the .properties file with the filename the user specified through a `JOptionPane`. The `Properties` class inherits from `Hashtable`, therefore the `put` and `putAll` methods can be applied to a Properties object. Their use was avoided as they allow the caller to insert entries whose keys or values are not Strings. The `setProperty` method was employed instead as shown in the Code Fragment [line 08-11]. The `PropertiesHandler` class has been implemented separately for handling all the necessary and required functionality. For storing the properties firstly a properties object must be constructed which will contain all the key and element pairs (ex. NOM.min = 0).

Metrics and properties

metric	use?	weight	min	max
CBO	☑	1	0	7
CCN	☑	1	0	7
DIT	☑	1	0	7
LCOM	☑	1	0	7
RFC	☑	1	0	7
WMC	☑	1	0	7

FIGURE 10: METRICS AND PROPERTIES

```
01.          public static Properties extractPropertiesFromTable(JTable t) {
02.              String[] prop = {"used", "weight", "min", "max"},
03.              Properties tempProp = new Properties();
04.              for (int i=0; i<t.getRowCount(); i++) {
05.                  String name = (String)t.getValueAt(i, 0);
06.                  String value = t.getValueAt(i, 1).toString();
07.                  tempProp.setProperty(name + ".used", value);
08.                  tempProp.setProperty(name + ".weight", value);
09.                  tempProp.setProperty(name + ".min", value);
10.                  tempProp.setProperty(name + ".max", value);
11.              }
12.              return tempProp;
13.          }
```

Having the properties object constructed from the table (Fig. 10) as shown in the method below then by utilising `OutputStream` and the `Properties.store` method the properties are output in the specified filename.

savePropertiesToPropertyFile()

```
01.          Properties p; //Contains current properties
02.          OutputStream propOS = new FileOutputStream(fileName);
03.          p.store(propOS, "Properties File to the Test Application");
04.          propOS.close();
05.
06.
```

LOADING

For loading the properties files (.properties) a custom `JDialog` has been developed which automatically finds all the saved .properties files and loads them into a `JList` for enhancing the usability of the system and minimising errors and mistakes. The Dialog is displayed (Fig. 11) in the Figure on the right. Given the filename of the Properties file to load

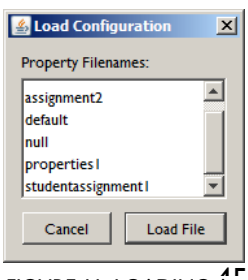


FIGURE 11: LOADING A PROPERTIES FILE

the class contains a `Properties.load` method that reads a property list (key and element pairs) from the `InputStream` and inputs them into the object. After the properties object is constructed with the loaded properties then the properties being displayed on the JTable (Fig. 10) are amended.

CONFIGURING

The configuration of the metrics when updated on the Properties table (Fig. 10) is achieved in the same way as discussed in §7.1.2 through the metric Accessor class. The Accessor class since is holds each of the metric files separately it also contains attributes for holding these properties of the metrics in order not the use again reflection apart from the initial load of the system. Therefore when a property is amended instead of updating the metric classes themselves -which would require more coding-, the accessor class is employed instead.

7.1.5 PHASE 5

The metrics analyse each of the classes individually and produce a measured value. This is achieved using the metric's Accessor class as discussed in Phase 2. Each of the metrics constructors has been implemented so it would be initialised and take as a parameter the class that will be evaluated. The metric's method is then run to evaluate the class and stores the measured value. The measured value is finally retrieved by accessing the getMeasuredValue method that return the measured value as an Integer. The metric accessor class dynamically invokes the methods using the `method.invoke(iClass)` method it returns an `Object` that can be parsed as a `String` and thereafter be converted to an `Integer`.

For each class every metric has been measured and results were stored in a 2D Array. The reason for creating a 2d Array it that the `JTable` used for displaying the results was setup to display the data using a the `AbstractTableModel`, an abstract class that provides default implementation for most of the methods in the `TableModel` interface. By using the `AbstractTableModel` the columns and the data that will be displayed are initialised when the table is created and therefore the complications associated with manually setting the data and columns names has been avoided.

7.1.6 PHASE 6

As discussed, metrics variate in the way they measure the classes. As an example measuring NOM a class shouldn't have a large number of classes and therefore when the value is measured the lower it is the better and as it increases the resulting mark should be lower.
For the normalisation of the measured values normalisation the `Normalisation` Class has been developed especially for this puprose and has been build to be indepented from other classes. The normalisation class retrieves for each of the metrics its measured value and its boundaries and returns the normalised value. Despite of numerous attempts a simple way for

normalising values except could not be found and the possible cases found were difficult to comprehend and actually use. Therefore as a result the min-max normalisation has been employed that can be calculated using the following formula:

$$MV = \frac{m - min}{max - min} (targetmax - targetmin) + targetmin$$

The formula is used to linearly transform the measured values in the [targetmin = 0, targetmax = 100] range, where the min and max represent the specified metrics boundary values. In case the measured value is greated than the targetmax boundary this immediately results as a targetmax, and in case its less that the targetmin it results as targetmin. Moreover since the metrics the higher their measured values are the less the resulting mark should be, the normalisation class inverts the values, meaning where NOM measured and normalised value is 20 the mark is then inverted to 80 in the specified [0, 100] range.

Finaly during this phase, the final mark will be calculated through the formula specified in §4.3 using the weights and finally displayed to the user.

7.1.7 IMPLEMENTING THE FRAMEWORK

Applied and combined the layers and finalisation of the tool. Applying the MVC design pattern to provide separation in the system's components and manage the communications and interconnection between the three main architecture layer; the model, view and the control. Each of the layers was contained within a package and a driver class was built to initialise the three layers. The implementation of the system using MVC design pattern was not an easy task as it has its complication because without a good design it would be difficult to separate the interaction of the components between the layers. However, the analysis and design performed before hand has proven to provide the essentials for having a basis to rely upon.

```
01.         public class driver {
02.             //... Create model, view, and controller.  They are
03.             //    created once here and passed to the parts that
04.             //    need them so there is only one copy of each.
05.             public static void main(String[] args) {
06.                 model      m   =  new model();
07.                 view       v   =  new view(m);
08.                 controller c   =  new controller(m, v);
09.
10.                 v.setVisible(true);   // display GUI
11.             }
12.         }
13.
```

MODEL: The model does not depend on the other layers. The model layer is composed of the main `model` class and several others that implement the functionality for measuring the classes (see Appendix J). The main reason for separating the model class is the decomposition of the layer. Tightly coupled classes are known to be difficult to extend and modify and having this in mind it was separated to the utmost extend.

VIEW: The view knows of the model since the view request from the model to assign and implement `Event Listeners` for each of the controls on the view and the actions associated with those controls when triggered. The GUI was developed based on the Interface designed in the analysis (see Appendix F).

CONTROL: The control now, based on the User Actions from the User Interface as it handles the GUI controls actions, makes appropriate requests to the model and if necessary returns the results from the model.

7.1.8 SYSTEM VALIDATIONS

As a final phase of the implementation was the addition of validations for minimising errors and features for improving the usability of the system. This section describes the features and validations added to the system.

For loading java classes or java projects, Java's `FileChosser` has been used and a custom `FileFilter` has been applied for only displaying .java files and the java files have associated .class files. Loading saved metrics properties is achieved through a custom dialog that has been implemented in order to restrict the users' actions and remove any possibilities of errors.
When saving a properties file the system checks if the file already exists and requests the user whether the file should be replaces. Another validation is added to the system is to ensure that already a project or a class is classes loaded to be able to run the evaluation.

HIGHLIGHTS AND CONCLUSIONS

The implementation of the system has been described and emphasis has been put on certain details of the implementation that were considered to be as the most important ones for each of the development phases. All the phases have been completed successfully and any issues and problems faced during the implementation have been tackled during the process. The metrics have been implemented and a Graphical user Interface has been constructed to enhance the system's usability.

TESTING PROCESS

This chapter describes how the testing of the system and in particular the metrics has been achieved. Mainly the methodology for testing the system was Use Case based testing, an approach where test cases are generated from the Use Cases defined in the requirements specification. The chapter also discusses the testing executed during the development of the phases and issues that have been identified and solved.

8.1 PHASE/FEATURE TESTING

At the end of each of the phase's implementation, informal tests have been performed to ensure that the features that have been implemented had the expected functionality and no errors emerged. Especially it was important to test the metrics whilst they've been developed to ensure the maximum precision possible in the measurements and overall functionality of the system. Numerous errors and bugs were solved in the attempt to deliver a fully operational and stable system, some of which are listed.

METRICS TESTING

For particular cases such as the CCN metric, a testing class has been constructed shown in Appendix M and the class's *bytecode* generated has been retrieved through the BCEL library and printed out (also shown in Appendix M) for matching and identifying the names of the instructions that will be measured. Particularly as it can be observed from Appendix M it was apparent which instructions will be measured: the switch, while, for, if, statements are of type `BranchInstruction` and the try...catch statement of type `Instruction`. Therefore by retrieving for each method's instructions in a class we look if any of the instructions match the decisions that increase Complexity.

Another issue identified whilst testing was loss of precision when calculating LCOM3. The problem was that whilst performing the calculating the returned value was automatically rounded and no decimal points where returned. After several tests the issue was successfully identified to be that despite that the attribute holding the measured value was a double, if the attributes used to perform that arithmetic operation were `Integer` then this resulted in this precision loss. Therefore all the attributes were changed to be declared as `Double` to avoid and fix this issue with losing precision in the results. Another issue identified, when calculating LCOM3, if a class has no methods, then this results in division by zero which throws an Exception. Therefore for all the metrics this needed to be handled and a condition to check the divided value has been set.

FUNCTIONAL TESTING

Moreover, tests have been performed that the classes are loaded correctly, the metrics are loaded and displayed correctly, the metric properties can be amended a new project can be loaded, single java files can be loaded, etc. Whenever or whilst a feature of the system was developed tests followed for ensuring their correctness and stability.

Generating manually Graphical User Interfaces with Java is known to be a taunting task since several problems might arise. One particular one was that after interacting with the system and clicking on the results table or resize the system caused the `centerPanel` to disappear. This problem was tackled by using a `JSplitPane` for separating the different areas, which also provides better functionality since the user can resize the panels.
Further tests were performed based on the Use Cases which will be demonstrated in the following section.

8.2 USE CASES TESTING

For testing purposes the testing method used is Use Case Based Testing. Test cases that are derived from use cases take advantage of the existing specification to ensure good functional test coverage of the system. Therefore the following table has been constructed displaying the test case performed that originated from the defined Use Cases in §5.2.1. For each of the test cases a description is provided along with the necessary preconditions that must be applicable for the correct functionality of the tool.

Testing the system revealed no unexpected errors as shown in the Table in Appendix Q. All the necessary actions of the system passed the tests and overall it can be said that the system is error-free in taking into account as well to the assumptions listed in §5.4.

HIGHLIGHTS AND CONCLUSIONS

During testing the system a lot of issues have been identified which have been successfully solved. From the final Use Case based testing no errors emerged and that the system can run effectively without any issues based on the preconditions specified.

HYPOTHESIS TEST

This chapter tests the hypothesis by testing that the system development meets some evaluation criteria being specified. This chapter discusses the choice of criteria, how they were measured, and the results. This chapter more specifically describes the choice of experimental method and the results of those experiments. Finally the results are analysed, including some consideration of their statistical significance and reliability.

HYPOTHESIS

"The development of a system, that will utilise metrics for automatically assessing the code quality, could provide accurate formative feedback which will reflect on the code's design quality."

Justifying the hypothesis of this project was the development of a system that by employing certain Object-Oriented metrics would assess the quality of students assignment's and measure the overall Object-Oriented design quality of the programming assignment based on how well the object-oriented concepts and mechanisms (see §1.2). To be able to answer whether this was a working hypothesis, meaning that provisionally this hypothesis is accepted by answering certain criteria. The criteria to be answered for testing whether the hypothesis can be characterised as working are:

1. *Does the system provide the availability of having different frameworks for achieving measuring various different projects?*
   At universities, from personal experience mainly, different projects are assigned to the students where each of them has different requirements and each of them expects students to apply particular different concepts of object-oriented programming where each one of them should be applied at a certain level. It is required from the system to provide this flexibility and availability of different frameworks in order not to restrict the availability of using the system in only certain projects and also be able to justify the degree of the object-oriented concepts or mechanisms applicability in the assignment.

2. *Is the feedback accurate enough and reliable for depicting the Object-Oriented quality of a student's assignment?*
   However, the main criterion for testing the hypothesis is the feedback accurateness as generated by the system. The system is required to provide an overall resulting mark that reflects the quality of the Java code based on the frameworks and applicability of object-oriented mechanisms discussed above.

   The testing of the hypothesis will rely on testing these criteria, and measuring how the system performs and what availability does it provide, if any.

HYPOTHESIS TESTING

9.1 MARKING FRAMEWORKS

Evaluating and testing the first criterion for reaching to a working hypothesis, the system successfully dynamically loads the metrics and has the ability to dynamically invoke the metrics for retrieving the results. This is also achievable for both a single class and a project as a whole. Tutors are provided to availability of evaluating single classes of a project are the whole project.

The system also provides the availability of configuring the system and the metrics. This results in achieving flexibility and therefore offers the Tutors the option of selecting which metrics they require for evaluating the student's assignment. The selection can be made by the tutor as each of the metrics maps to an object-oriented concept of mechanism, constructing therefore a custom framework for evaluating the code quality of the assignment.

To establish control over the metrics measurements, minimum and maximum boundaries have been set for each of the metrics. These boundaries can be configured allowing this way the tutors to raise or lower the boundaries in order to achieve better results in the measurement and calculation of the final mark.
Moreover the system allows Tutors choosing the degree that the metrics should count when calculating the final mark, the weight. This provides the option of selecting appropriate weights for the concepts to be evaluated and set the degree that the metric would count towards the final mark.

These features of the tool functionality have been tested thoroughly during the testing phase (see §8). Hence as a result this criterion was met successfully. The second criterion set was however the most important. The next section discusses how testing and evaluating the successfulness of the criterion has been achieved.

9.2 SYSTEM FEEDBACK

The system from the previous criterion evaluation showed that it can actually dynamically load the metric class, statically analyse the code and provide the measurements. However, still, the main important criterion that the system needs to fulfil in order for to be able to characterise the hypothesis successful is the accuracy of the feedback provided from the system. To achieve testing this criterion an experiment has to be set up to take measurements over a test project and compare with the actual code whether the results depict the actual object-oriented quality both overall as the final mark and more specifically the metrics measured values.

9.2.1 EXPERIMENT

The experiment needed to be set-up to help verify the hypothesis that this system could be actually used for marking students' assignments. For this an example project by Bjork (2004) has been identified as it meets the criteria, its relatively well decomposed and structured and in general makes good use of the object-oriented concepts. Bjork (2004) is a tutor teaching Object-Oriented development and particularly developed this ATM project example for demonstrating a complete OO methodology applied to a single problem. Proceeding further, the ATM project will be analysed using the system developed to test the accuracy of the results through a series of experiments.

The experiments' that will be performed is measurement of the code and modification of the metric properties (Boundaries, Weights) and compare the results. Main purpose of this experimental analysis is not to validate only whether these metrics can provide reliable results since this has been proven by several authors in the literature, but mainly to investigate whether or not the metrics have been successfully been implemented and the overall mark can actually represent a reliable result that the tutors can find dependable and representative of the actual code quality.

9.2.2 RESULTS

The experiment was conducted by specifying the minimum, maximum boundaries based on the one identified as most appropriate in the literature (see §5.1.2). The ATM project has been evaluated and the results are displayed in the table below:

METRIC	BOUNDARIES MIN	BOUNDARIES MAX	MEASURED AVERAGE	WEIGHT	MARK
AHF	0	7	93.5	100	0.9346798
CBO	0	10	66.2	100	0.6621212
DIT	0	7	100	100	1
LCOM3	1	2	65.3	100	0.6531057
MHF	0	7	99.7	100	0.9969899
NOA	0	7	59.3	100	0.592803
NOM	0	10	82.6	100	0.8257576
RFC	0	18	100	100	1
TCC	0	10	100	100	1
				FINAL MARK	**79**

TABLE 4 – EXPERIMENTAL RESULTS A

From the final mark and the average for each of the metrics not a lot of feedback can be gained. Mainly as it can be observed the resulting final mark, 79% presents a very good overall result for the project which conceptually depicts the actual quality of the project. The measured values after being normalised are displayed in Appendix P. For each of the classes and each of the metrics a measured value is displayed. This can help particularly in finding which classes lack of object oriented quality. To investigate this for each of the metrics a class that has the minimum, a medium, and the maximum mark have been randomly selected and manually assessed the code and compared to evaluate how effective these results are.

METRIC	CLASSES	MEASURE (Rounded)	Notes
AHF	*Status*	100	All attributes are declared as private
	Message	91	5 Public and 7 private methods
	Card	85	one public method
CBO	*AccountInformation*	100	No coupling to other classes
	NetworkToBank	50	Coupled to Simulation, Log and InetAddress
	Simulation	0	Initialises the simulated components of the ATM GUI Coupled to 10 classes
DIT	*ATM*	100	No inheritance generally in the whole project
LCOM3	*Card*	100	Only one attribute is accessed only by one method
	SimReceiptPrinter	75	Two attributes accessed by two methods could not be considered as cohesive
	Message	0	This class's LCOM is 0 due to the get/set methods accessing the local variables.
MHF	*SimulatedBank*	100	All methods are private(5) except one
	SimulatedBank$InvalidPin	75	All methods (2) are public
	SimulatedBank$Failure	50	All methods (5) are public
RFC	*Deposit*	87	No responses
	CardReader	50	3 methods 3 responses made
	Transaction	0	More than 8 responses
TCC	*GUI*	100	No decisions made
	CustomerConsole	70	4 decisions made in 6 methods
	Message	0	8 decisions made, exceeds the max boundary
NOM	*AccountInformation*	100	no methods
	SimulatedBank	30	9 methods
	Simulation	0	19 methods
NOA	*SimOperatorPanel*	100	no attributes defined
	Receipt	62	3 attributes
	ATM	0	18 attributes

TABLE 5 – CRITICAL VALIDATION OF METRIC RESULTS

Overall the results from the Table 5 above appear to depict the actual quality of the code, but in some cases such as LCOM the results appear to be to some extent erroneous. Moreover one fact is that classes associated with the GUI interface tend to have high AHF, CBO, LCOM but low MHF, TCC and RFC values which ends up giving a balance between the two therefore it is not noticeable on the final mark.

One of the problems noticeably is the way the measured values have been normalised. NOM normally should be normalised based on a function similar to the sigmoid function's shape where from 0...8 the *mark=100%* since a certain number of attributes in a class is acceptable, and after reaching the maximum point, 8, the resulting mark decreases downward until the maximum boundary limit. This is achievable with the tool by setting the minimum boundary to 8, so any lower values than the boundary are acceptable and hence would result in scoring 100%. Therefore from the results table shown above the resulting mark for *SimulatedBank* for the NOM metric should be ~90 instead of 30 where it's more acceptable and sensible as a result since it contains only 9 methods. In similar cases a class coupling to

1-2 classes might be acceptable therefore by raising the boundaries the overall results would improve significantly. This derived in the conclusion that to attain and achieve best results the system has to be configured as appropriate. This is not necessarily a drawback since this provides the tutors to raise and lower the boundaries as appropriate and based on their own personal criteria when marking.

The main problem though lies elsewhere. Knowing how the metrics work and achieve the measurements it can be said that the system can be easily misled and deceived. This is not a case for all the metrics. This is not something that can be handled easily and managing these deficiencies would be very complicated task. To explain the problem more clearly taking as an example AHF and MHF, these metrics look at the accessibility of a class's the attributes and methods have respectively. A class that has 5 attributes 1 of which is hidden has AHF=hidden/total=0.2. However the system does not take into account whether these attributes are actually a part of the class and referenced in the code or publically accessed from a different class. Now, if 5 more attributes are added as hidden, AHF becomes 0.6 giving therefore a better result. Therefore due to lack of formal specification, the actual formulas for calculating the metrics and the incomplete specification of the AHF given by Al-Ja'Afer and Sabri (2005), the feedback from AHF, MHF cannot be actually included for marking. Therefore the system needs to be measured at the system level for certain metrics and certain validations need to be made to produce more reliable and efficient results.

Concluding from the above, some of the metrics specifications need to be altered. Specifically AHF and MHF specification should be altered to include that the Attributes/Methods need to be actually referenced in an outside class to be counted towards the calculation of the metrics. Moreover the system requires to be configured accordingly for each project to achieve the required measurements and reliability.

9.3 TESTING CONCLUSIONS

From the second criterion for testing the hypothesis two words are highlighted in the question: *reliable*, and *accurate*.

Based on the experiments executed the results of the hypothesis testing it have been proven for certain metrics that the way they have been implemented does not allow the utmost possible *accurateness and reliability* in the measurement thus they have been considered as inappropriate to be utilised for marking student's assignments. The system cannot be used in isolation but still can generate measurements that could help tutors get a good idea of the object-oriented quality of the code in hand, by looking at the project's classes after using the system for the analysis. The metrics developed have been widely acknowledged by the authors in the literature and therefore their results can be dependable but as discovered the system can be misled by the code. As a result further validations need to be made in the way metrics achieve their measurements to be able to rely solely on the system's results.

EVALUATION

The evaluation chapter evaluates the quality of the project as a whole, including a consideration of how well each of the objectives was met. The chapter discusses the effectiveness of the literature review, design, implementation, testing, the techniques used through the development and alternative hypotheses that could have been tested. For the testing the chapter discusses the experimental results and the conclusions reliability.

EVALUATION OF PROJECT

LITERATURE REVIEW

The literature review was one of the most time consuming parts of the whole project because of three main reasons, the overwhelming number of articles available in the area of metrics and the metrics currently available and finally the unavailability of established frameworks for marking. The review started by analysing the object-oriented concepts and mechanisms for later to base the selection of metrics within those certain aspects. By further commenting on the research currently has been done in the field and looking into particular evaluations the research was focused into certain suites restricting the number of metrics to investigate even less. Finally by establishing a framework certain metrics then have been identified that deemed appropriate for measuring those aspects of object-oriented programming.

Since the way of approaching the project was not established initially this lead to the above problems, took more time and the research was not performed as efficiently as possible. Another approach which would have been more preferable was the set-up interviews to collect and establish a common marking framework from tutors in the University. This however has not impacted the project negatively since this lead upon taking the decision of making the system more flexible to allow the selection of metrics to be used in the code evaluation.

The literature has proved to be effective enough to identify metrics that deemed as appropriate for the system that has been developed. However as discussed above no information of established marking schemes exists, and therefore improvisation was required of to achieve the desired results. This lead construction of a scheme for classifying and choosing the most appropriate metrics. Evidently, the literature review was in depth for a certain extent, but it was limited to the results established from other authors, which results are adequate enough for reliance but on the other hand any further investigation would increase the size of the project and most probably the project could not be delivered within the time constraints.

ANALYSIS/SPECIFICATION

The metric specification as presented through examples helped both in the design and development of the metrics but also makes cleared for the readers to understand how the particular metrics work and achieve the measurements. Moreover by specifying the metrics properties, boundaries and weights initially everything was considered and plotted before hand and therefore the development later became easier and straightforward. However for the metrics normalisation some possible ways of achieving that have been identified, such as the ones proposed by Xie, et al. (2000). These however have been proven very difficult to understant and even more complicate to realise since knowledge in Statistics was required. Trying to find a way for normalising the metrics has taken an excessive ammount of time but the attempts were proven ineffective.

The analysis and requirements section (§5.2) helped in specifying the requirements of the products. This phase of the project was one of the most successful one since correct decisions were made from the start and the overall collection of the requirements has been very thorough and carefully listed providing a comprehensive list afterwards to follow when proceeding with the design and implementation of the system. The initial analysis of the requirements was performed using Use Case analysis which aided in identifying the requirements and therefore construct the list of requirements (see §5.2.2). Another important mechanism that contributed to the successful and on time delivery of the system was the prioritisation technique, Moscow rules (see §5.3.3). Despite that this is simple technique yet it is very powerful since priorities have been set over which requirements, functions and metrics of the system and have been developed in a prioritised manner, and therefore had better control over the project.

The system was developed using the Iterative methodology (see §5.3.1). Utilisation of this methodology proved to be more than helpful in the system's development since each of the requirements was designed, implemented and tested in isolation. This resulted in better designed objects but relatively slowed down the development since at the end the objects had to be combined in the final phase to compose the final system.

DESIGN

The design is another phase of the project that can be defined as being successful. Critical decisions have been taken during this phase that had a vital impact on the project on time delivery and successfulness. The analysis of possible approaches for certain requirements revealed the most appropriate techniques for implementing those requirements. The decisions were taken upon the time availability, the simplicity of the approaches and finally the required functionality obtainable. Hence the approaches selected were simple enough to be developed within the available time and were expected to provide the required functionality.

Further the design chapter describes how UML was exploited and defined the system requirements in models to achieve breaking up and decomposing the system easier, in order to divide its complexity into discrete pieces that were studied individually. Two distinct diagrams have been developed; a class and a dataflow diagram each with its own distinct characteristics and purpose. This

UML representation of the system provided the infrastructure for the system to be developed and consequently the development process progressed fast and effectively.

IMPLEMENTATION

From one point of view, the implementation phase in general was successful since the system was completed successfully but still on the other hand it's not unblemished. The system was design and implemented based on a structured framework therefore structurally it's considerably well implemented but further refinement and decomposition of the main classes could be achieved.
The decision was taken to create a Graphical User Interface (GUI) for the system instead of just a text based one. Creating the interface, adding the listeners and handling the interaction between the MVC components took a lot of time which could be have been spend on improving other parts of the system and therefore have more time to spend on particular phases. However, on the other hand the GUI enhanced significantly the usability of the system which could apparently increase the desirability of using the system. The reason for developing the system on the first hand is for simplifying the way tutors would mark assignments and provide an even better way more reliable for measuring the object-oriented quality.

Any significant details of the development have been described in the report and a general overview of the metrics development. Particular details on the metrics implementation have not been included mainly for limiting the size of the report. Further discussion on the technical details of the system is provided in the next chapter that evaluated the build quality of the product.

TESTING

The approach used for performing the testing was not the best since Use Case based testing is mostly shallow and does not test the inner parts of the system. Despite that, UCBT provided adequate enough feedback for characterising the system as functional, however further tested would be most appropriate. Such methodologies are Black-box and Unit testing, that can actually test a system to its limits and as thoroughly as possible, when utilised correctly. Black-box could have been used to perform boundary value analysis to test what the results are where the user has input to the system and more specifically changing the metrics properties. Unit testing could help is performing a complete and exceptionally thorough testing of the system by analysing the system in individual units of source code as for each method a different test would be written to ensure that code meets its design and behaves as intended.

HYPOTHESIS TESTING

The results showed that a project can be evaluated efficiently and gain accurate results about a system's object-oriented quality. Despite that the methodology used cannot be said to be the most accurate and the most sufficient one. The accurately evaluate the successfulness of the system designed and testing the hypothesis the experiment should have been performed using actual data from student's assignments which have already been marked by tutors. Comparing then both results the ones from the tutors and the systems the comparative analysis would provide a more definite

and reliable results. This was unfortunately not possible due to that it was not planned initially and had no access to the data needed.

PROJECT ORGANISATION AND PLANNING

Planning was evidently as discussed above in the analysis of the overall process, inadequate. Particular phases such as the hypothesis testing and the literature survey lacked initial analysis and therefore when the development plan was constructed misjudging the requirements resulted in time scarcity. The literature review for the project consumed most of the time for the reason denoted above. This resulted in getting off-track from the project plan constructed in the Terms of reference (see Appendix R). It was important though to revise the plan for managing that the system would be delivered on the deadline. Evidently a lot of time that would be spend on other tasks was lost and therefore the development of the other phases needed to be constricted and done in less time that originally planned. However the project was finished successfully on time, a week before allowing time to revise, make the final changes and hand-in.

PERSONAL GAINS

It can be said that a lot has been gained throughout the project's development but mainly knowledge and experience. It was very intriguing and interesting researching the field of metrics since their applicability is very broad. Most importantly when personally developing applications the system developed can be used, to identify flaws in the system being developed and tackled early in the development stage. Experience was gained in using Reflection which has gained a lot of interest because of the flexibility it provides, using architectural frameworks such as MVC that is widely used in the industry and overall had the chance to improve and exercise both personal academic and practical skills.

EVALUATION OF PRODUCT

The product is characterised by two main features, its build quality and fitness for purpose of which will be evaluate to have an overall and complete evaluation of the product. The build quality will be described by technically evaluating the parts that compose the system and the build quality of the system at both the framework and the code level. The fitness for purpose will be derived from the hypothesis testing and evaluation of the results to define whether the hypothesis has proven as working.

11.1 PRODUCT BUILD QUALITY

This chapter describes technically the quality of the system that has been developed. First of all starting from the overall quality, the decomposition of the system is good, since the employment of the *Model-View-Controller pattern* aided the breakdown of the whole system into finer more controllable parts. Further could be established use of command pattern for the controller part of the system. The model, view, controller classes handling the functionality of the system therefore the size of the classes could be even further broken down and decomposed into more classes.

Considering the Controller, it could be decomposed using the *Command pattern* as this is exactly what the controller purpose is, retrieve the user commands and call appropriate methods. Instead thought of having methods objects of the different commands would be needed which would implement a `Command` interface.

Java Collections have been used throughout the system, even for the metrics implementation. Collections offer extensibility since their size does not need to be defined initially. Besides this, it is not the most distinct characteristic that Java Collections have to offer. `ArrayLists` allow duplicates and have been used for storing the metrics objects and the java classes loaded in the system. The reason for this is that a java class might be contained in the same project in a different package with the same name thus, it should not be discarded. `Sets` and `Maps` discard any duplicate data. These were used throughout the metrics implementation in cases were duplicated needed to be discarded. In particular, when measuring coupling, a reference in the class of another class should only be counted once despite the number of times referenced, hence any duplicates need to be discarded. Good use has been made over Java's Collections throughout the project which resulted in having less code to write and a more effective way of realising the functionality.

Taken as a whole, the required functionality has been offered and successfully been implemented to even provide a graphical user interface. The GUI provided better usability, made the system faster to use, provided better ways of displaying results and the ability of viewing the measured classes for performing criticism on the evaluation results.

11.2 PRODUCT FITNESS FOR PURPOSE

The system can be characterised as fit for its purpose only if the hypothesis has been proven successfully. The system achieved to provide Tutors an alternative for marking but still the system is not without its flaws. To achieve correct marking the metrics have to be selected and configured appropriately. Therefore it is achievable with the expense of required configuration.

The main issue thought is as identified the specifications of the metrics and the way they have been implemented as discussed in the hypothesis testing (§9.3). Second issue is the normalisation technique used. Using linear normalisation is not the appropriate solution since the marks should not decrease linearly as the measured value increases in all cases. Taking as example DIT, no inheritance means a high mark but as it increases the mark for DIT decreases, where the mark should have remained high until the measured value reached the breakpoint (the average of the min, max boundaries of the metric).

Solely the tool cannot be used in isolation for marking purposes. The system can give some very good results and evidently provide accurate results on the code quality for finding erroneous and not maintainable classes. It could be a very good system that can be employed to look into the code characteristics and therefore the tutors can attain particular information and insight on the Object-Oriented quality of the students' assignments. Hence the tutors can indentify classes and specific parts of the system that have not been designed well according to the metrics analysis results.

CONCLUSIONS

Overall, the project success in apparent, both the aims and objectives of the project have been met and executed successfully. The following is a discussion of the conclusions drawn throughout the project and how these map to the requirements as defined by the aims and objectives of the project.

AIMS

For the project two main aims have been initially specified of which the conclusions of each will be listed:

"The key aim of this project is to identify, categorise and choose Object-Oriented Metrics that could be utilised for assessing the design quality of students programming assignments"

A set of the metrics (§3.3.3), was put together by various metrics and specifically *NOA*, *NOM*, *TCC*, *DIT*, *LCOM3*, *RFC*, *CBO*, AHF, *MHF* has been successfully identified through the report based on validations performed by researchers in the field. It has been concluded that manual assessment of code has several issues associated (§1.3.1). Therefore the need for automatic assessment is apparent and has been emphasized by many authors in the field (§1.3.2). It has been identified that several systems have been proposed of which none specifically implemented for particular use in assessing student assignments (§1.3.2). For achieving the development of such a system appropriate metrics had to be selected. The selection relied on the concepts and mechanisms of the object-oriented paradigm that when applied correctly define the quality of systems. For the main concepts of object-oriented programming, coupling, cohesion, information hiding, inheritance and class objects, methods and attributes (§1.2) accordingly appropriate metrics have been defined. The metric set defined, can be characterised as inclusive since as it gives a full and comprehensive coverage of the object-oriented aspects (see §3.3) apart from polymorphism which had been identified as complicated to implement and therefore had a lower priority in the development (§5.3.3). Hence, this aim can be said that it has been processed and achieved in success.

"To design, implement, test an automatic assessment tool that can use Object-Oriented metrics for evaluating Java code's design for specified design quality attributes which can be used to provide informative feedback to the user."

The second aim now was the application of the metrics chosen for this purpose and development of a system that automatically analyses the code and returns the overall mark representing the quality of the code. The system has been successfully designed §6, implemented §7 and tested §8 using an iterative methodology composed of phases (see §6.4), where for each phase one specific requirement of the system has been developed. All the requirements specified have been successfully implemented. From the hypothesis testing the quality of the feedback gained from the system has been analysed. The results showed that the in sum the system provided very good results but it was not without its problems (§9.2.2 & §9.3), since some metrics have been identified that can be inaccurate which was an issue that concerned their specification and appropriate changes have been

noted (§9.2.2). Another important conclusion was that, to attain reliable results the system has to be configured as appropriate (§9.2.2). The system provides this flexibility of adjusting the boundaries so the tutors can raise and lower the boundaries as appropriate to attain the desirable results. Evidently the project in general was a success and important conclusions have been drawn of how the system can be improved, therefore it can be said that this aim has been carried out successfully.

OBJECTIVES

The objectives defined for the project were as follow:

- To review the problems involved when manually assessing software's quality and how automatic assessment would resolve those. (§1.3.1)
- To review how using metrics could be used in assessing the software's quality (§1.3.2)
- To identify and review the metrics available for automated assessment (§2 & §4)
- To select criteria and evaluate the metrics identified (§3)
- To design and implement a tool that will automatically assess Java classes based on the selection of required metrics. The tool will provide an overall mark which indicates the design quality of the class being assessed. (§5, §6 & §7)
- To test the project's hypothesis (§8)
- To evaluate the project and the project's success (§10)

These objectives have been successfully met as discussed in the conclusions and the discussion above. Each of the objectives maps to a section or chapter included in this document therefore it can be said that each one of them is actually depicted and actually met in the project.

FUTURE WORK

Many ideas of possible future work that could be done in the field arose throughout the development of the project. One possible future implementation is performing run-time analysis for the evaluation instead of static analysis that currently is used. Moreover, the addition of new metrics could be considered and implementation of the ones that have been omitted such as metrics for measuring Polymorphism. An addition to the tool can be considered to improve the feedback provided by the system. For example, for coupling the system could should to which classes a class is coupled to provide consequently more reasonable and detailed results to the user.

One important future work that has been identified in the literature is the lack of marking frameworks in the literature. Therefore a project could be devoted in identifying and establishing marking frameworks by performing questionnaires to tutors and identifying particular aspects that are being measured and looked at when assessing code. This could derive in the creation of new metrics tailored to the requirements for assessing the quality of students' code and could be incorporated in the current tool.

Finally, SourceForge.net is the world's largest open source software development web site that provides free services that help people build software and share it globally. The current prototype can be uploaded on the site and be made available for other programmers to extend and modify.

REFERENCES

Al-Ja'Afer, J. J., and Sabri, K.E. (2004) 'Chidamber-Kemerer (CK) and Lorenze-Kidd (LK) metrics to assess Java Programs'.

Al-Ja'Afer, J. J., and Sabri, K.E. (2005) 'Metrics for Object-Oriented Design (MOOD) to assess Java Programs'.

Ala-Mutka, K. M. (2005) 'A survey of automatic assessment approaches for programming assignments', *Comput. Sci. Edu.,* 15 (2), pp. 83-102.

Amandeep, K., Satwinder, S. and Kahlon K.S. (2009) 'Evaluation and Metrication of Object Oriented System', *Proceedings of the International MultiConference of Engineers and Computer Scientists 2009,* I, pp. 1063-1068 *IMECS 2009* [Online]. Available at: http://www.iaeng.org/publication/IMECS2009/IMECS2009_pp1063-1068.pdf (Accessed: 11 March 2010).

Basili, R. V., Briand, L. and Melo, L.W. (1995) 'A Validation of Object-Oriented Design Metrics as Quality Indicators', *IEEE Transactions on Software Engineering,* 22 (10), pp. 751 - 761 *ACM* [Online]. Available at: http://portal.acm.org/citation.cfm?id=239308 (Accessed: 11 March 2010).

Benlarbi, S., and Melo, W. L. (1999a) 'Polymorphism measures for early risk prediction', *In Proceedings of the 21st international Conference on Software Engineering ICSE '99*, pp. 334-344 *ACM* [Online]. Available at: http://doi.acm.org/10.1145/302405.302652 (Accessed: 11 March 2010).

Benlarbi, S., and Melo, W.L. (1999b) 'Polymorphism Measures for Early Risk Prediction', *21st International Conference of Software Engineering (ICSE'99)*, pp. 334-344.

Benlarbi, S., El-Emam, K., Goel, M., and Rai, S. (1999) (1999c) 'Thresholds for object-oriented measures', *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE'00)*, p. 24 *IEEE Computer Society*.

Benlarbi, S., Emam, E.K., and Goel, N. (1999d) 'Issues in Validating Object-Oriented Metrics for Early Risk Prediction', *FastAbstract ISSRE*, pp. 1-2 [Online]. Available at: http://www.cistel.com/free_expertise/publications/Cistel-1999-10.pdf (Accessed: 11 March 2010).

Bjork, R. C. (2004) *ATM Simulation* [Computer Program]. [Online]. Available at: http://www.math-cs.gordon.edu/courses/cs211/ATMExample/ (Accessed.

Booch, G. (1994) *Object-Oriented Analysis and Design with Applications*. 2nd edn edn. Benjamin Cummings.

Briand , L., Daly, W., and Wust, J (1999) 'A Unified Framework for Coupling Measurement in Object-Oriented Systems', *IEEE Transactions on Software Engineering,* 25, pp. pp.91-121.

Briand, L. C., Daly, J. W., and Wuest, J. (1997) 'A Unified Framework for Cohesion Measurement', *In Proceedings of the 4th international Symposium on Software Metrics (November 05 - 07, 1997)*, p. 43 *IEEE Computer Society* [Online]. Available at: http://www.computer.org/portal/web/csdl/doi/10.1109/METRIC.1997.637164 (Accessed: 11 March 2010).

Briand, L. C., Wüst, J., Daly, J., and Porter, V (1998) 'A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems', *In Proceedings of the 5th international Symposium on Software Metrics (March 20 -*

21, 1998), p. 246 *IEEE Computer Society* [Online]. Available at: http://portal.acm.org/citation.cfm?id=823920# (Accessed: 11 March 2010).

Brito e Abreu, F. (2001) 'Using OCL to formalize object oriented metrics definitions', [Online]. Available at: http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/others/MOOD_OCL.pdf (Accessed.

Brito e Abreu, F., and Carapuça, R. (1994a) 'Candidate metrics for object-oriented software within a taxonomy framework', *J. Syst. Softw.,* 26 (1), pp. 87-96 [Online]. Available at: http://dx.doi.org/10.1016/0164-1212(94)90099-X (Accessed: 11 March 2010).

Brito e Abreu, F., and Carapuça, R. (1994b) 'Object-Oriented Software Engineering: Measuring and Controlling the Development Process', *4th Internation Cond. On Software Quality, CiteSeerX* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.1382&rep=rep1&type=pdf (Accessed.

Brito, F. (1995) 'Talk on "Design Metrics for Object-Oriented Software Systems"', *ERCIM Workshop Proceedings,* 95 (W001)*ERCIM* [Online]. Available at: http://www.ercim.eu/publication/ws-proceedings/7th-EDRG/7th-EDRG-contents.html (Accessed: 11 March 2009).

Brooks, C., and Buell, C. (1994) 'A Tool For Automatically Gathering Object-Oriented Metrics', *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference (NAECON 1994),* 2, pp. 835-838 [Online]. Available at: http://www.comp.nus.edu.sg/~bimlesh/oometrics/8/00332952.pdf (Accessed: 11 March 2010).

Brooks, I. (1993) 'Object-Oriented metrics collection and evaluation with a software process', *OOPSLA '93 Workshop on Processes and Metrics for Object-Oriented Software Development,* 26.

Bruntink, M., and Deursen, A. (2004) 'Predicting Class Testability using Object-Oriented Metrics', *In Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE international Workshop (September 15 - 16, 2004)*, pp. 136-145 *SCAM. IEEE Computer Society, Washington, DC* [Online]. Available at: http://dx.doi.org/10.1109/SCAM.2004.15 (Accessed: 11 March 2010).

Brusilovsky, P., and Higgins, C. (2005) 'Brusilovsky, P. and Higgins, C.', *ACM Journal of Educational Resources in Computing,* 5 (3).

Cartwright, M., and Shepperd, M. (2000) 'An Empirical Investigation of an Object-Oriented Software System', *IEEE Trans. Softw. Eng.,* 26 (8), pp. 786-796 [Online]. Available at: http://dx.doi.org/10.1109/32.879814 (Accessed: 11 March 2009).

Cau, A., Concas, G. and Marchesi, M. (2006) 'Extending OpenBRR with Automated Metrics to Measure Object Oriented Open Source Project Success', *Workshop on Evaluation Frameworks for Open Source Software (EFOSS) Como*, pp. 1-4 [Online]. Available at: http://www.openbrr.org/como-workshop/papers/CauConcasMarchesi_EFOSS06.pdf (Accessed: 11 March 2010).

Cheang, B., Kurnia, A., Lim, A., and Oon, W. (2003) ') On automated grading of programming assignments in an academic institution', , *Computers & Education,* 41, pp. 121-131.

Chidamber, R. S., Darcy, P.D., and Kemerer, F.C. (1998) 'Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis', *IEEE Transactions on Software Engineering,* 24 (8), pp. 629-639 [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=34E92EB5A38C15817E1EE5427A9311E1?doi=10.1.1.61.2217&rep=rep1&type=pdf (Accessed: 11 March 2010).

Chidamber, S. R., and Kemerer, C.F. (1994) 'A Metrics Suite for Object Oriented Design', *IEEE Trans. Software Eng.,* 20 (6), pp. 476-493 [Online]. Available at: http://dx.doi.org/10.1109/32.295895 (Accessed: 11 March 2010).

Churcher, N., and Irwin, W. (2003) 'Object Oriented Metrics: Precision Tools and Configurable Visualisations', [Online]. Available at: http://ir.canterbury.ac.nz//handle/10092/3028 (Accessed: 11 March 2010).

Ciupke, O. (1999) 'Automatic Detection of Design Problems in Object-Oriented Reengineering', *In Proceedings of the Technology of Object-Oriented Languages and Systems (August 01 - 05, 1999)*, p. 18 *IEEE Computer Society,*

Washington, DC [Online]. Available at: http://portal.acm.org/citation.cfm?id=833062# (Accessed: 11 March 2010).

Clúa, O., and Feldgen, M. (2008) 'Work in Progress - Object Oriented Metrics and Programmers' Misconceptions', *38th ASEE/IEEE Frontiers in Education Conference*, pp. F3H-19-F13H-20 *IEEE* [Online]. Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=04720377 (Accessed: 11 March 2010).

Deligiannis, I., Sfetsos, P., Stamelos, I., Angelis, L. Xatzigeorgiou, R., and Katsaros, P. (no date) 'Assessing the Modifiability of two Object- Oriented Design Alternatives - A Controlled Experiment Replication ', [Online]. Available at: http://delab.csd.auth.gr/~katsaros/Experimental-Replication.pdf (Accessed: 11 March 2009).

Emam, E. K. (2001) 'A Primer on Object-Oriented Measurement', *Seventh International Software Metrics Symposium (METRICS'01)*, pp. 1-3 *IEEE* [Online]. Available at: http://www.computer.org/portal/web/csdl/doi/10.1109/METRIC.2001.915527 (Accessed: 11 March 2010).

Fenton, N. E., and Pfleeger, S.L. (2004) *Metrics: A Rigorous and Practical Approach* 2nd Revisited Printing edn. London: International Thomson Computer Press.

Forsythe, G. E., and Wirth, N. (1965) 'Automatic Grading Programs', *Communications of ACM,* 8 (5), pp. pp.275-529 [Online]. Available at: ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/65/17/CS-TR-65-17.pdf (Accessed.

Glasberg, D., El-Emam, K., Melo, W. and Madhavji, N. (2000) 'Validating Object-Oriented Design Metrics on a Commercial Java Application', *National Research Council 44146,* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.1103&rep=rep1&type=pdf (Accessed: 11 March 2010).

Gupta, S. B. (1997) 'A Critique of Cohesion Measures in the Object-Oriented Paradigm', *Michigan Technological University, Department of Computer Science*.

Guyomarc'h, J. Y., and Gueheneu, Y.G. (no date) 'On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics', [Online]. Available at: http://www.iro.umontreal.ca/~sahraouh/qaoose2005/paper4.pdf (Accessed: 11 March 2010).

Harisson, R., Counsell, S., and Nithi, R. (1997) 'An Overview of Object-Oriented Metrics', *In Proceedings of the 8th international Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, p. 230 *IEEE Computer Society* [Online]. Available at: http://portal.acm.org/citation.cfm?id=831977# (Accessed: 11 March 2010).

Harrison, R., Counsell, S. J., and Nithi, R. V. (1998a) 'An Evaluation of the MOOD Set of Object-Oriented Software Metrics', *IEEE Transactions on Software Engineering,* 24, pp. 491-496 [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.5269 (Accessed: 11 March 2010).

Harrison, R., Counsell, S.J. and Nithi, R.V. (1998b) 'An Evaluation of MOOD set of Object-Oriented Software Metrics', *IEEE Trans. Software Engineering,* SE-24 (6), pp. PP.491-496.

Henderson-sellers, B. (1996) *Object-Oriented Metrics, Measures of Complexity*. Prentice Hall.

Hext, J. B., And Winings, J.W. (1969) 'An automatic grading scheme for simple programming exercises', *ACM,* 12 (5), pp. pp.272-275 [Online]. Available at: http://portal.acm.org/citation.cfm?id=362981 (Accessed.

Hollingsworth, J. (1960) 'Automatic graders for programming classes', *Communications of ACM 3,* 10, pp. 528-529.

Hung, S., Lam-For, S., and Raymond, C. (1992) 'Automatic Program Assessment', *Automatic Program Assessment,* 20 (2), pp. 183-190

ISO (2001) *ISO/IEC 9126-1: Software engineering Product quality: Quality model.* . [Online]. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749 (Accessed: 20 June 2010).

Jamali, S. M. (2006) 'Object Oriented Metrics. (A Survey Approach)', [Online]. Available at: http://ce.sharif.edu/~m_jamali/resources/ObjectOrientedMetrics.pdf (Accessed: 11 March 2010).

Kanmani, S., Sankaranarayanan, V., and Thambidurai, P. (2006a) 'Evaluation of Object Oriented Metrics', [Online]. Available at: http://www.ieindia.org/pdf/86/pcn5fl4.pdf (Accessed: 11 March 2006).

Kanmani, S., Uthariaraj, V.R., Sankaranarayanan, V., and Thambidurai, P. (2006b) 'Object-Oriented software fault prediction using neural networks', *Information Software Technology,* 49, pp. 438-492.

Karimi, J., and Konsynski, B.R. (1988) "An automated software design assistant', *IEEE Transaction on Software Engineering,* 14, pp. 194-210.

Karunanithi, S., and Bieman, M.J. (1992) 'Candidate Reuse Metrics For Object Oriented and Ada Software', *Proc. IEEE International Software Metrics Symposium*, pp. 120-128 [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.6703&rep=rep1&type=pdf (Accessed: 11 March 2010).

Khan, R. A., Mustafa, K. and Ahson, S.I. (2007) 'An Empirical Validation of Object Oriented Design Quality Metrics', *J. King Saud Univ.,* 19, pp. 1-16 *Comp. & Info. Sci.* [Online]. Available at: http://colleges.ksu.edu.sa/ComputerSciences/Documents/Paper%201-16.pdf (Accessed: 11 March 2010).

Koten, V. C., and Gray, A. (2005) 'An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability', *Information and Software Technology,* 48 (1), pp. 59-67 *Science Direct* [Online]. Available at: http://www.sciencedirect.com/ (Accessed: 11 March 2010).

Lanza, M., and Ducasse, S. (no date) 'Beyond Language Independent Object-Oriented Metrics: Model Independent Metrics', [Online]. Available at: http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Lan-Duc.pdf (Accessed: 11 March 2010).

Lee, Y., Liang, B., Wu, S., and Wang, F. (1995) 'Measuring the Coupling and Cohesion of an Object-Oriented program based on Information flow', *Proc. Int'l Conf. Software Quality*.

Li, W., and Henry, S. (1993) 'Object-oriented metrics that predict maintainability', *J. Syst. Softw.,* 23 (2), pp. 111-122 [Online]. Available at: http://dx.doi.org/10.1016/0164-1212(93)90077-B (Accessed: 11 March 2010).

Lindholm, T., and Yellin, F. (no date) *The JavaTM Virtual Machine Specification.* 2nd edn [Online]. Available at: http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html (Accessed.

Marinescu, R. (1998) 'Using Object-Oriented Metrics for Automatic Design Flaws Detection in Large Scale Systems', *In Workshop Ion on Object-Oriented Technology (July 20 - 24, 1998) S. Demeyer and J. Bosch, Eds. Lecture Notes In Computer Science,* 1543, pp. 252-255 *Springer-Verlag, London,* [Online]. Available at: http://portal.acm.org/citation.cfm?id=704663# (Accessed: 11 March 2010).

Marinescu, R. (2005) 'Measurement and Quality in Object-Oriented Design', *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 1-4 *IEEE* [Online]. Available at: http://loose.upt.ro/download/papers/radum-phd-icsm05.pdf (Accessed: 11 March 2010).

Martin, R. (1995) 'OO Design Quality Metrics: An Analysis of Dependencies', *ROAD,* 2 (3) [Online]. Available at: http://www.objectmentor.com/resources/articles/oodmetrc.pdf (Accessed: 11 March 2010).

McCabe , J. T. (1976) 'A Complexity Measure', *IEEE Transactions on Software Engineering,* 2 (4), p. 308 [Online]. Available at: http://www.literateprogramming.com/mccabe.pdf (Accessed: 11 March 2010).

McCabe, T. J., Dreyer, L.A., Dunn, A.J., and Watchon, A.H. (1994) 'Testing an object-oriented application', *J. Quality Assurance Institute*, pp. pp.21-27.

McConnel, S. (2004) *Code Complete*. 2nd edn. edn. Redmond, WA: Microsoft Press.

McConnell, S. (2004) *Code Complete: A Practical Handbook of Software Construction*. 2nd edn edn.

Menger, A. S., and Ulans, J.V. (1999) 'A Case Study of the Analysis of Novice Student Programs', *CSEE&T*, pp. 40-49.

Ort, E., and Mehta, B. (2003) *Java Architecture for XML Binding (JAXB), SDN.* [Online]. Available at: http://java.sun.com/developer/technicalArticles/WebServices/jaxb/ (Accessed: 10 June 2010).

Pereira, S. R., and Csar, F.E. (2008) 'Metrics to Evaluate the Use of Object Oriented Frameworks', *Computer Journal,* 52 (3), pp. 288-304(217) *Oxford University Press* [Online]. Available at: http://www.ingentaconnect.com/content/oup/cj/2009/00000052/00000003/art00003 (Accessed: 11 March 2010).

Redish, K., and Smyth, W. (1986) 'Style Analysis: A Natural By-product of Program Compilation', *Communication of the ACM,* 29 (2), pp. 126-133.

Rosenberg, L. H. (1998) 'Applying and Interpreting Object Oriented Metrics', *Software Technology Conference,* [Online]. Available at: http://satc.gsfc.nasa.gov/metrics (Accessed: 11 March 2010).

Rosenberg, L. H., and Hyatt, L.E. (no date) 'Software Quality Metrics for Object-Oriented Environments', pp. 1-6 [Online]. Available at: http://www.cetin.net.cn/storage/cetin2/QRMS/ywxzqt24.htm (Accessed: 11 March 2009).

Salehie, M., Li, S., and Tahvildari, L. (2006) 'A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws', *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 159 - 168 *IEEE* [Online]. Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1631118 (Accessed: 11 March 2010).

Sarkar, S., Kak, C.A., and Rama, G.M. (2008) 'Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software', *IEEE Transactions on Software Engineering,* 24 (5), pp. 700-720 [Online]. Available at: http://dx.doi.org/10.1109/TSE.2008.43 (Accessed: 11 March 2010).

Schroeder, M. (1999) 'A Practical Guide to Object-Oriented Metrics', *IT Professional 1, 6 (Nov. 1999)*, pp. 30-36 [Online]. Available at: http://dx.doi.org/10.1109/6294.806902 (Accessed: 11 March 2010).

Sherif, J. S., and Sanderson, P. (1998) 'Metrics for object-oriented software projects', *J. Syst. Softw.,* 44 (2), pp. 147-154 *ACM* [Online]. Available at: http://dx.doi.org/10.1016/S0164-1212(98)10051-1 (Accessed: 11 March 2010).

Subramanian, G. a. C., W. (2001) 'An empirical study of certain object-oriented software metrics ', *Journal of Systems and Software,* 59 (1), pp. 57-63 [Online]. Available at: http://portal.acm.org/citation.cfm?id=507999.508003 (Accessed: 11 March 2010).

Subramanyam, R., and Krishnan, M.S. (2003) 'Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects', *IEEE Transactions on Software Engineering,* 29 (4), pp. 297-310 [Online]. Available at: http://portal.acm.org/citation.cfm?id=766682 (Accessed: 11 March 2010).

Talbi, T., Meyer, B. and Stapf, E. (2001) 'A metric framework for object-oriented development', *39th International Conference of Object-Oriented Languages and Systems (TOOLS 39),* (TOOL), pp. 164-172 [Online]. Available at: http://portal.acm.org/citation.cfm?id=884718 (Accessed: 11 March 2010).

Tang, M., Kao, M., and Chen, M. (1999) 'An Empirical Study on Object-Oriented Metrics', *Proceedings of the 6th international Symposium on Software Metrics*, p. 242 *IEEE Computer Society* [Online]. Available at: http://portal.acm.org/citation.cfm?id=520792.823979 (Accessed: 11 March 2010).

Tegarden, P. D., Sheetz, S.D. and Monarchi, E.D. (1996) 'Effectiveness of Traditional Software Metrics for Object-Oriented Systems', *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences,* 4, pp. 359-368 [Online]. Available at: http://www.acis.pamplin.vt.edu/faculty/tegarden/wrk-pap/HICSS25.PDF (Accessed: 10 March 2010).

Vaishnavi, V. K., Purao, S., and Liegle, J. (2006) 'Object-oriented product metrics: A generic framework', *Information Sciences,* 177, pp. 587–606 [Online]. Available at:

http://iris.nyit.edu/~kkhoo/Spring2008/Topics/DS/OOProdMetrics_InfoSci-Vaishnavi-Purao-Liegle-2007.pdf (Accessed: 11 March 2010).

Wakil, E. M., and Fahmy, A. (no date) 'Software Metrics - A Taxonomy ', *Faculty of Computers and Information,* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7086&rep=rep1&type=pdf (Accessed: 11 March 2010).

Wakil, E. M., Bastawisi, E.A., Boshra, M. and Fahmy, A. (no date) 'Object-Oriented Design Quality Models A Survey and Comparison ', *Citeseer* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.1254&rep=rep1&type=pdf (Accessed: 11 March 2010).

Xenos, M., Stavrinoudis, D., Zikouli, K. and Christodoulakis, D. (2000) 'Object-Oriented Metrics - A Survey', *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000*, pp. 1-10 [Online]. Available at: http://citeseer.ist.psu.edu/528212.html (Accessed: 11 March 2010).

Xenos, M. a. C., D. (1997) 'Measuring Perceived Software Quality', *Information and Software Technology,* 39 (5), pp. 417-424 [Online]. Available at: http://dx.doi.org/10.1016/S0950-5849(96)01154-8 (Accessed: 11 March 2010).

Xie, T., Yuan, W., Mei, H., and Yang, F. (2000) 'JBOOMT: Jade Bird Object-Oriented Metrics Tool', [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.1260 (Accessed: 11 March 2010).

Xu, J., Ho, D. and Capretz, L.F. (2008) 'An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction', *Journal of Computer Science,* 4 (7), pp. 571-577 *2008 Science Publications* [Online]. Available at: http://www.scipub.org/fulltext/jcs/jcs47571-577.pdf (Accessed: 11 March 2010).

APPENDICES

APPENDIX A: METRICS LIST (XENOS ET AL., 2000)

TRADITIONAL METRICS

name	description
AML (**A**verage **M**odule **L**ength)	measures the average module size
BAM (**B**inding **A**mong **M**odules)	measures data sharing among modules
CCN (**C**yclomatic **C**omplexity **N**umber)	measures the number of decisions in the control graph
CDF (**C**ontrol flow complexity and **D**ata **F**low complexity)	is a combine metric based on variable definitions and cross-references
COC (**C**onditions and **O**perations **C**ount)	counts pairs of all conditions and loops within the operations
COP (**Co**mplexity **P**air)	combines Cyclomatic complexity with logic structure
COR (**Co**upling **R**elation)	assigns a relation to every couple of modules according to the kind of coupling
CRM (**C**ohesion **R**atio **M**etrics)	measure the number of modules having functional cohesion divided by the total number of modules
DEC (**De**cision **C**ount)	offers a method to measure program complexity
DSI (**D**elivered **S**ource **I**nstructions)	counts separate statements on the same physical line as distinct and ignores comment lines
ERE (**E**xtent of **Re**use)	categorises a unit according the lever of reuse (modifications required)
ESM (**E**quivalent **S**ize **M**easure)	measures the percentage of modifications on a reused module
EST (**E**xecutable **St**atements)	counts separate statements on the same physical line as distinct and ignores comment lines, data declarations and headings
FCO (**F**unction **Co**unt)	measures the number of functions and the source lines in every function
FUP (**Fu**nction **P**oints)	measures the amount of functionality in a system
GLM (**Gl**obal **M**odularity)	describes global modularity in terms of several specific views of modularity
IFL (**I**nformation **Fl**ow)	measures the total level of information flow between individual modules and the rest of a system
KNM (**Kn**ot **M**easure)	is the total number of crossing points on control flow lines
LOC (**L**ines **O**f **C**ode)	measures the size of a module
LVA (**L**ive **Va**riables)	deals with the period each variable is used
MNP (**M**inimum **N**umber of **P**aths)	measures the minimum number of paths in a program and the reachability of any node
MOR (**Mor**phology metrics)	measure morphological characteristics of a module, such as size, depth, width and edge-to-node ratio
NLE (**N**esting **Le**vels)	measures the complexity as depth of nesting
SSC (composite metric of **S**oftware **S**cience and **C**yclomatic complexity)	combines software science metrics with McCabe's complexity measure
SSM (**S**oftware **S**cience **M**etrics)	are a set of composite size metrics
SWM (**S**pecification **W**eight **M**etrics)	measure the function primitives on a given data flow diagram
TRI (**Tr**ee **I**mpurity)	determines how far a graph deviates from being a tree
TRU (**Tr**ansfer **U**sage)	measures the logical structure of the program

OBJECT-ORIENTED METRICS

name	description/suite

CLASS METRICS

AHF (**A**ttribute **H**iding **F**actor)	is the ratio of the sum of inherited attributes in all system classes under consideration to the total number of available classes attributes (**MOOD**)
CCO (**C**lass **Co**hesion)	measures relations between classes

CEC (**C**lass **E**ntropy **C**omplexity)	measures the complexity of classes based on their information content
CLM (**C**omment **L**ines per **M**ethod)	measures the percentage of comments in methods
DAM (**D**ata **A**ccess **M**etric)	is the ratio of the number of private attributes to the total number of attributes declared in the class
FOC (**F**unction **O**riented **C**ode)	measures the percentage of non object–oriented code that is used in a program
INP (**In**ternal **P**rivacy)	refers to the use of accessory functions even within a class
LCM (**L**ack of **C**ohesion between **M**ethods)	indicates the level of cohesion between the methods (**Chidamber and Kemerer Suite**)
MAA (**M**easure of **A**ttribute **A**bstraction)	is the ratio of the number of attributes inherited by a class to the total number of attributes in the class
MFA (**M**easure of **F**unctional **A**bstraction)	is the ratio of the number of methods inherited by a class to the total number of methods accessible by members in the class
MHF (**M**ethod **H**iding **F**actor)	is defined as the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration (**MOOD**)
NAD (**N**umber of **A**bstract **D**ata types)	is the number of user-defined objects used as attributes in a class that are necessary to instantiate an object instance of the class
NCM (**N**umber of **C**lass **M**ethods in a class)	measures the measures available in a class but not in its instances
NIV (**N**umber of **I**nstance **V**ariables in a class)	measures relations of a class with other objects of the program
NOA (**N**umber **O**f **A**ncestors)	is the total number of ancestors of a class
NPA (**N**umber of **P**ublic **A**ttributes)	counts the number of attributes declared as public in a class
NPM (**N**umber of **P**arameters per **M**ethod)	**NPM** (**N**umber of **P**arameters per **M**ethod)
NRA (**N**umber of **R**eference **A**ttributes)	counts the number of pointers and references used as attributes in a class
PCM (**P**ercentage of **C**ommented **M**ethods)	is the percentage of commented methods
PDA (**P**ublic **Da**ta)	counts the accesses of public and protected data of a class
PMR (Percent of **P**otential **M**ethod uses actually **R**eused)	is the percentage of the actual method uses
PPD (**P**ercentage of **P**ublic **D**ata)	is the percentage of the public data of a class
RFC (**R**esponse **F**or a **C**lass)	is the number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class (**Chidamber and Kemerer Suite**)
WCS (**W**eighted **C**lass **S**ize)	is the number of ancestors plus the total class method size
WMC (**W**eighted **M**ethods per **C**lass)	is the sum of the weights of all the class methods (**Chidamber and Kemerer Suite**)

METHOD METRICS

AMC (**A**verage **M**ethod **C**omplexity)	is the sum of the Cyclomatic complexity of all methods divided by the total number of methods
AMS (**A**verage **M**ethod **S**ize)	measures the average size of program methods
MAG (**MA**X V(**G**))	is the maximum Cyclomatic complexity of the methods of one class
MCX (**M**ethod **C**omple**x**ity)	relates complexity with the number of messages

COUPLING METRICS

CBO (**C**oupling **B**etween **O**bjects)	counts the number of classes a class is coupled with (**Chidamber and Kemerer Suite**)
CCP (**C**lass **Cou**pling)	measures connections between classes based on the messages they exchange
CFA (**C**oupling **Fa**ctor)	is the ratio of the maximum possible number of couplings in the system to the actual number of couplings not imputable to inheritance (**MOOD**)

INHERITANCE METRICS

AIF (**A**ttribute **I**nheritance **F**actor)	is the ratio of the sum of inherited attributes in all classes of the system under consideration to the total number of available attributes for all classes (**MOOD**)
DIT (**D**epth of **I**nheritance **T**ree)	measures the number of ancestors of a class (**Chidamber and Kemerer Suite**)
FEF (**F**actoring **Ef**fectiveness)	is the number of unique methods divided by the total number of methods
FIN (**F**AN-**IN**)	is the number of classes from which a class is derived and high values indicate excessive use of

	multiple inheritance
HNL (Class **H**ierarchy **N**esting **L**evel)	measures the depth in hierarchy that every class is located
MIF (**M**ethod **I**nheritance **F**actor)	is the ratio of the sum of the inherited methods in all classes to the total number of available methods for all classes (**MOOD**)
MRE (**M**ethod **Re**use metrics)	indicate the level of methods reuse
NMI (**N**umber of **M**ethods **I**nherited)	measures the number of methods a class inherits
NMO (**N**umber of **M**ethods **O**verridden)	is the number of methods need to be re-declared by the inheriting class
NOC (**N**umber **O**f **C**hildren)	is the total number of children of a class (**Chidamber and Kemerer Suite**)
PFA (**P**olymorphism **Fa**ctor)	is the ratio of the actual number of possible different polymorphic situations of a class to the maximum number of possible distinct polymorphic situations for this class (**MOOD**)
PMO (Percent of **P**otential **M**ethod uses **O**verridden)	is the percentage of the overridden methods
RDB (**R**atio between **D**epth and **B**readth)	is the ratio between the depth and the width of the hierarchy of the classes
RER (**Re**use **R**atio)	is the ratio of the number of superclasses divided by the total number of classes
SIX (**S**pecialisation **I**nde**x**)	measures the type of specialisation
SPR (**Sp**ecialisation **R**atio)	is the ratio of the number of subclasses divided by the number of superclasses

SYSTEM METRICS

ADI (**A**verage **D**epth of **I**nheritance)	is computed by dividing the sum of nesting levels of all classes by the number of classes
ANA (**A**verage **N**umber of **A**ncestors)	determines the average number of ancestors of all the classes
APG (**Ap**plication **G**ranularity)	is the total number of objects divided by the total number of function points
ASC (**As**sociation **C**omplexity)	measures the complexity of the association structure of a system
CAN (**Ca**tegory **N**aming)	divides classes into semantically meaningful sets
CRE (Number of time a **C**lass is **Re**used)	measures the references in a class and the number of the applications that reuse this class
FDE (**F**unctional **De**nsity)	is the ratio of LOC to the function points
NCT (**N**umber of **C**lasses **T**hrown away)	measures the number of times a class is rejected until it is finally accepted
NOH (**N**umber **O**f **H**ierarchies)	is the number of distinct hierarchies of the system
OLE (**O**bject **L**ibrary **E**ffectiveness)	is the ratio of the total number of object reuses divided by the total number of library objects
PRC (**P**roblem **R**eports per **C**lass)	measures defect reports on this class
PRO (**P**ercent of **R**eused **O**bjects Modified)	declares the percentage of the reused objects that have been modified
SRE (**S**ystem **Re**use)	declares the percentage of the reuse of classes

APPENDIX B: METRICS LIST (KANMANI ET AL., 2006)

COUPLING MEASURES

CBO : (Coupling Between Objects) Number of other classes to which this class is coupled. A class is coupled to another if methods of one class use methods or attributes of the other or vice versa .
CBO' : Same as CBO except that inheritance is accounted.
RFC : (Response set for a Class) Number of methods in the response set of the class.
RFC-1 : Same as RFC except that methods indirectly invoked are not included.
MPC : (Message Passing Coupling) The number of method invocations in a class.
DAC : (Data Abstract Coupling) the number of attributes in a class that have as their type another class.5
DAC' : This count the unique classes used in the attributes.5
ICP : (Information Flow Coupling) Number of method invocations weighted by number of parameters of the invoked methods.
IH-ICP : (Inheritance Based Information flow Coupling) Count of invocations of methods of ancestors of classes only.
NIH-ICP : (Non-Inheritance Based Information Flow Coupling) Count of invocations to classes not related through inheritance).7
IC : (Inheritance Coupling) Number of parent classes to which a class is coupled.
COF : (Coupling Factor) Percentage of pairs of classes that are coupled.

For the following, the first one/two alphabets represent the relationship, the next two alphabets refer to the type of interaction and the last two refers to the impact of interaction.
IFCAIC : (Inverse Friend Class Attribute Import Coupling)
ACAIC : (Ancestor Class Attribute Import Coupling)
OCAIC : (Other Class Attribute Import Coupling)
FCAEC : (Friend Class Attribute Export Coupling)
DCAEC : (Descendants Class Attribute Export Coupling)
OCAEC : (Other Class Attribute Export Coupling)
IFCMIC : (Inverse Friend Class Method Import Coupling)
ACMIC : (Ancestor Class Method Import Coupling)
OCMIC : (Other Class Method Import Coupling)
FCMEC : (Friend Class Method Export Coupling)
DCMEC : (Descendant Class Method Export Coupling)
OCMEC : (Other Class Method Export Coupling)
IFMMIC : (Inverse Friend Method Method Import Coupling)
AMMIC : (Ancestor Method Method Import Coupling)
OMMIC : (Other Method Method Import Coupling)
FMMEC : (Friend Method Method Export Coupling)
DMMEC : (Descendant Method-Method Export Coupling)
OMMEC : (Other Method Method Export Coupling)

COHESION MEASURES

LCOM1 : (Lack of Cohesion in Methods) Number of pairs of methods in the class using no attribute in common.

LCOM2 : (Lack of Cohesion in Methods) Number of pairs of methods in the class using no attributes in common - Number of pairs of methods that do. (if difference is negative, LCOM2 is set to 0)
LCOM3 : (Lack of Cohesion in Methods) Number of connected components in the graph. Class is represented as an undirected graph with methods of the class as vertices and edges (if two methods use at least an attribute in common)
LCOM4 : (Lack of Cohesion in Methods) Similar to LCOM3 - it additionally includes an edge if method m invokes n or vice versa.
LCOM5 : (Lack of Cohesion in Methods) (1/a(Sμ(Aj))-m)/(1-m) - μ(Aj) is the number of methods(Mi,i=1,…m) with reference Aj(Aj,j=1,…a)
CO : (Connectivity)= 2((¡E¡-(¡V¡-1))/((¡V¡-1)(¡V¡-2))1
Coh : a variation on LCOM5: (Sμ(Aj)-m)/(ma)1
TCC : (Tight Class Cohesion) Pairs of methods which indirectly or directly use common attributes.
LCC : (Loose Class Cohesion) same as TCC but accounts methods indirectly connected.
ICH : (Information Flow-based Cohesion) Sum of the ICH (number of invocations of other methods of the same class weighted by the number of parameters of the invoked method) values of the methods in the class.

INHERITANCE MEASURES

DIT : (Depth of Inheritance) Length of the longest path from the class to the root in the hierarchy.
AID : (Average Inheritance Depth of a class).3 AID (parent classes) +1 or 0 (for classes without parent)
CLD : (Class to Leaf Depth) Maximum number of levels in the hierarchy that are below the class.
NOC : (Number of Children) Number of classes directly inherited.
NOP : (Number of Parents) Number of classes that a given class directly inherits from.
NOD : (Number of Descendants) Number of classes directly/indirectly inherited from this class.
NOA : (Number of Ancestors) Number of classes from which the class is inherited (directly/indirectly).
NMO: (Number of Methods Overridden) Number of methods overrides the inherited method.
NMinh: (Number of Methods inherited) Number of methods in a class that the class inherits from its ancestors and does not override.
NMA : (Number of Methods Added) Number of methods not inherited not overridden.
SIX : (Specialization Index) NMO*DIT/(NMO+NMA+NMinh)
MIF : (Method Inheritance Factor) Percentage of methods that are inherited in the system.
AIF : (Attribute Inheritance Factor) Percentage of attributes that are inherited in the system.
POF : (Polymorphic Factor) Percentage of possible opportunities for the method overriding that are used.
SPA : (Static Polymorphism in Ancestors) Number of function members that implement the same operator in ancestors and in the current class (at compile time).
DPA : (Dynamic Polymorphism in Ancestors) Number of function members that implement the same operator in ancestors and in the current class (at run time).
SPD : (Static Polymorphism in Descendants) Number of function members that implement the same operator in descendants and in the current class (at compile time).
DPD : (Dynamic Polymorphism in Descendants) Number of function members that implement the same operator in descendants and in the current class (at run time).
SP : (Static Polymorphism) SPA+SPD
DP : (Dynamic Polymorphism) DPA+DPD
NIP : (Non-Inheritance Polymorphism)
OVO : (Overloading in standalone classes) Count of number of functions with the same name in a class

APPENDIX C: METRIC LIST (AMANDEEP ET AL., 2009)

CLASS LEVEL

Response for a Class (**RFC**) Class (Chidamber and Kemerer, 1994)
Number of Attributes per Class (**NOA**) Class (Henderson, 1996)
Number of Methods per Class (**NOM**) Class (Henderson, 1996)
Weighted Methods per Class (**WMC**) Class (Chidamber and Kemerer, 1994)
Coupling between Objects (**CBO**) Coupling (Chidamber and Kemerer, 1994)
Data Abstraction Coupling (**DAC**) Coupling (Henderson, 1996)
Message Passing Coupling (**MPC**) Coupling (Henderson, 1996)
Lack of Cohesion (**LCOM**) Cohesion (Chidamber and Kemerer, 1994)
Tight Class Cohesion (**TCC**) Cohesion (Briand et al., 1999)
Loose Class Cohesion (**LCC**) Cohesion (Briand et al., 1999)
Information based Cohesion (**ICH**) Cohesion (Lee, 1995)
Number of Children (**NOC**) Inheritance (Chidamber and Kemerer, 1994)
Depth of Inheritance (**DIT**) Inheritance (Chidamber and Kemerer, 1994)
Number of Methods Overridden by a subclass (**NMO**) Polymorphism (Henderson, 1996)

SYSTEM LEVEL

Coupling Factor (**CF**) Coupling (Brito and Carapuca, 1994)
Method Inheritance Factor (**MIF**) Inheritance (Brito and Carapuca, 1994)
Attribute Inheritance Factor (**AIF**) Inheritance (Brito and Carapuca, 1994)
Method Hiding Factor (**MHF**) Information Hiding (Brito and Carapuca, 1994)
Attribute Hiding Factor (**AHF**) Information Hiding (Brito and Carapuca, 1994)
Polymorphism Factor (**PF**) Polymorphism (Brito and Carapuca, 1994)
Reuse ratio Reuse (**RRR**) (Henderson, 1996)
Specialization ratio Reuse (**SRR**) (Henderson, 1996)

APPENDIX D: SELECTED LIST OF METRICS

TRADITIONAL

Cyclomatic Complexity Number (**CCN**)

OBJECT-ORIENTED

MOOSE (C.K.)

Depth of Inheritance Tree (**DIT**)
Coupling between object classes (**CBO**)
Response for a Class (**RFC**)
Lack of Cohesion of Methods (**LCOM**)

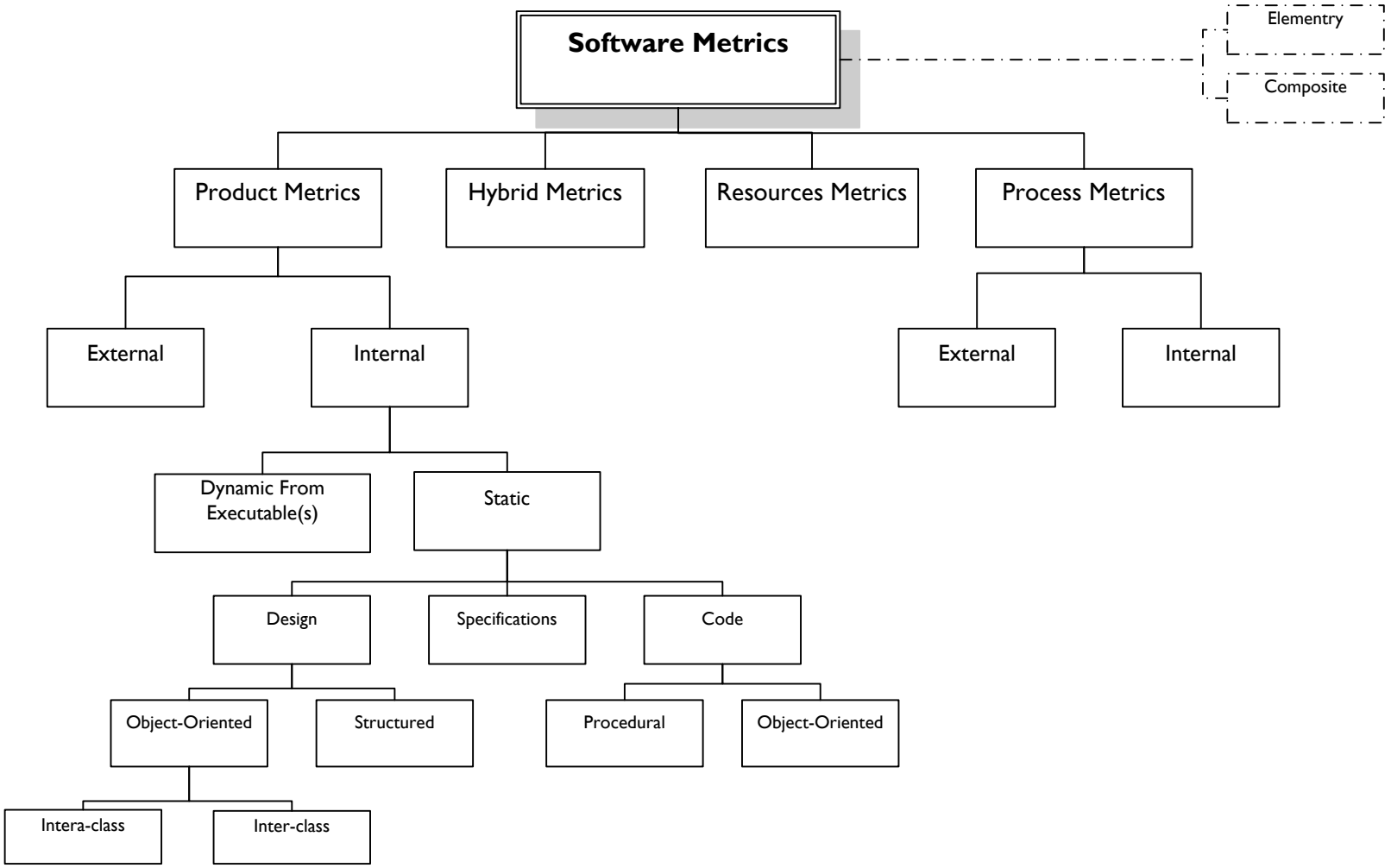
MOOD

Coupling Factor (**CF**)
Polymorphism Factor (**PF**)
Method Hiding Factor (**MHF**)
Attribute Hiding Factor (**AHF**)

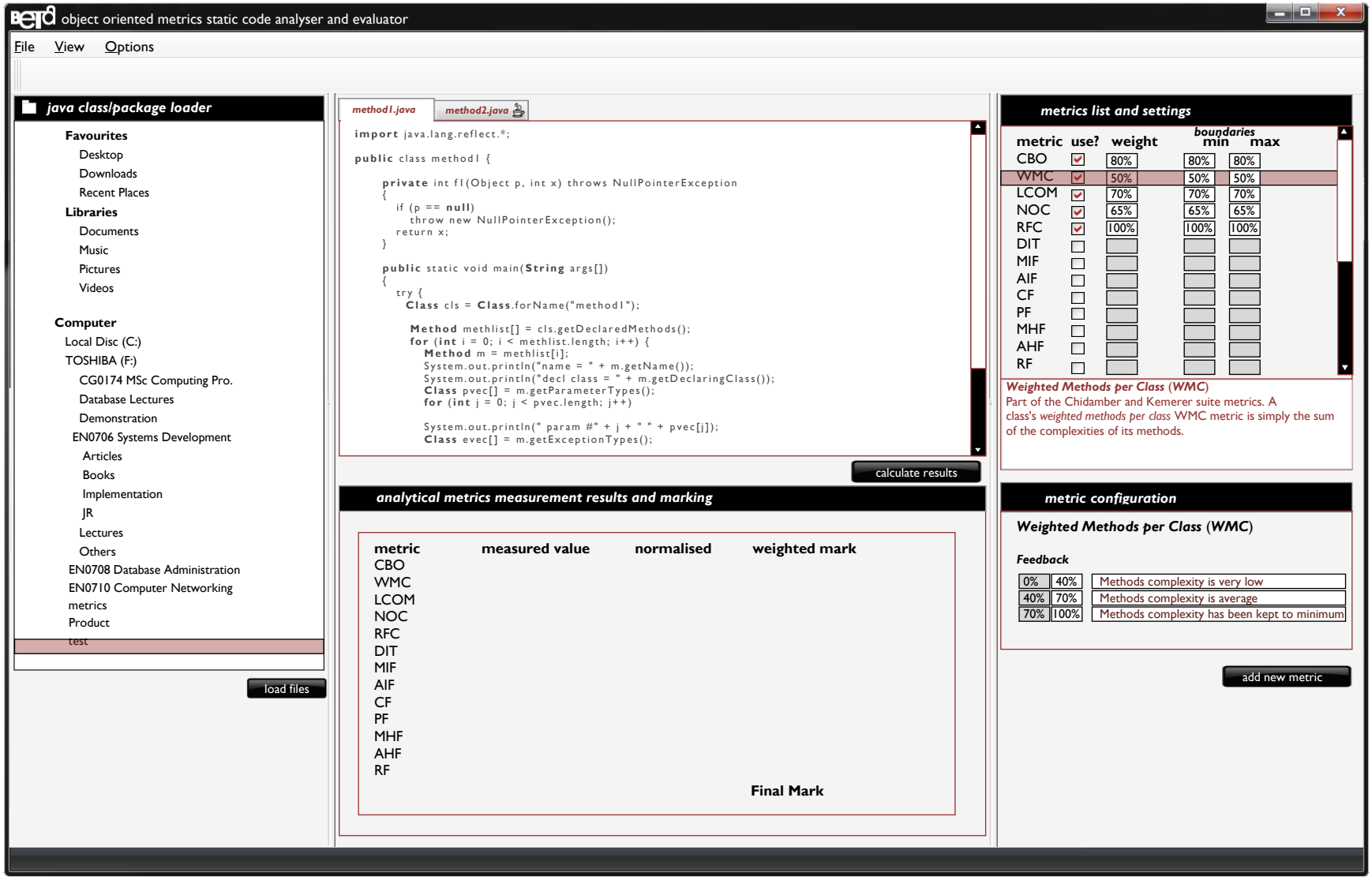
LORENZ & KIDD

Number of Attributes per Class (**NOA**)
Number of Methods per Class (**NOM**)

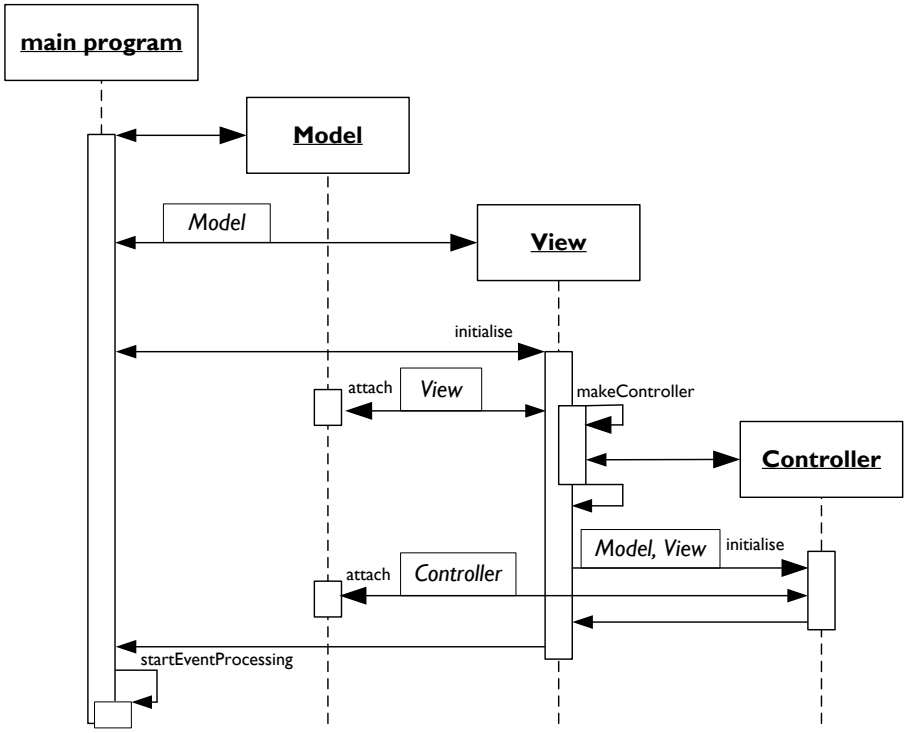
APPENDIX E: SOFTWARE METRICS TAXONOMY



APPENDIX F: USER INTERFACE DESIGN AND FUNCTIONALLITY

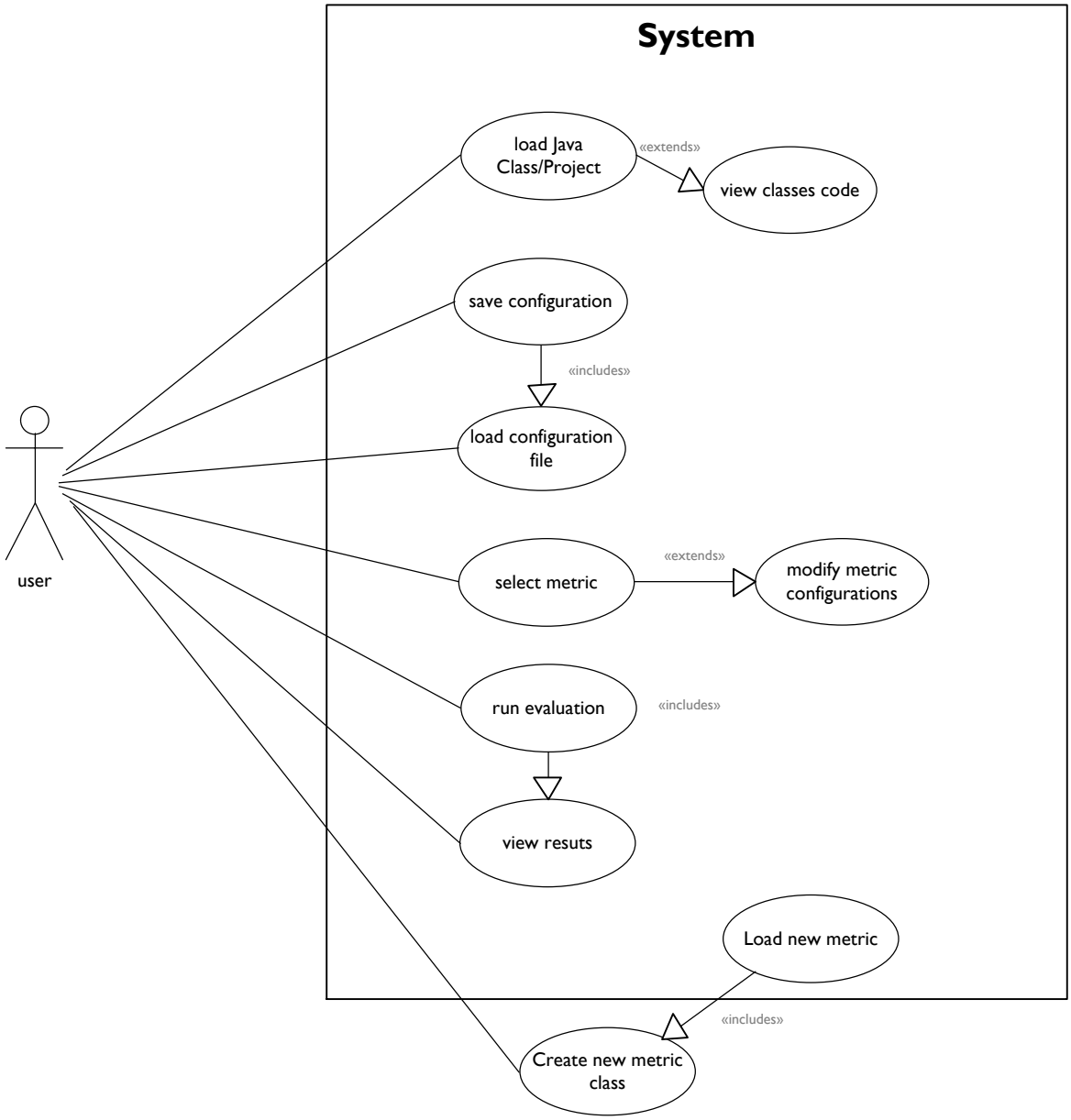


APPENDIX G: CONSTRUCTION OF OBJECTS OF MVC PATTERN

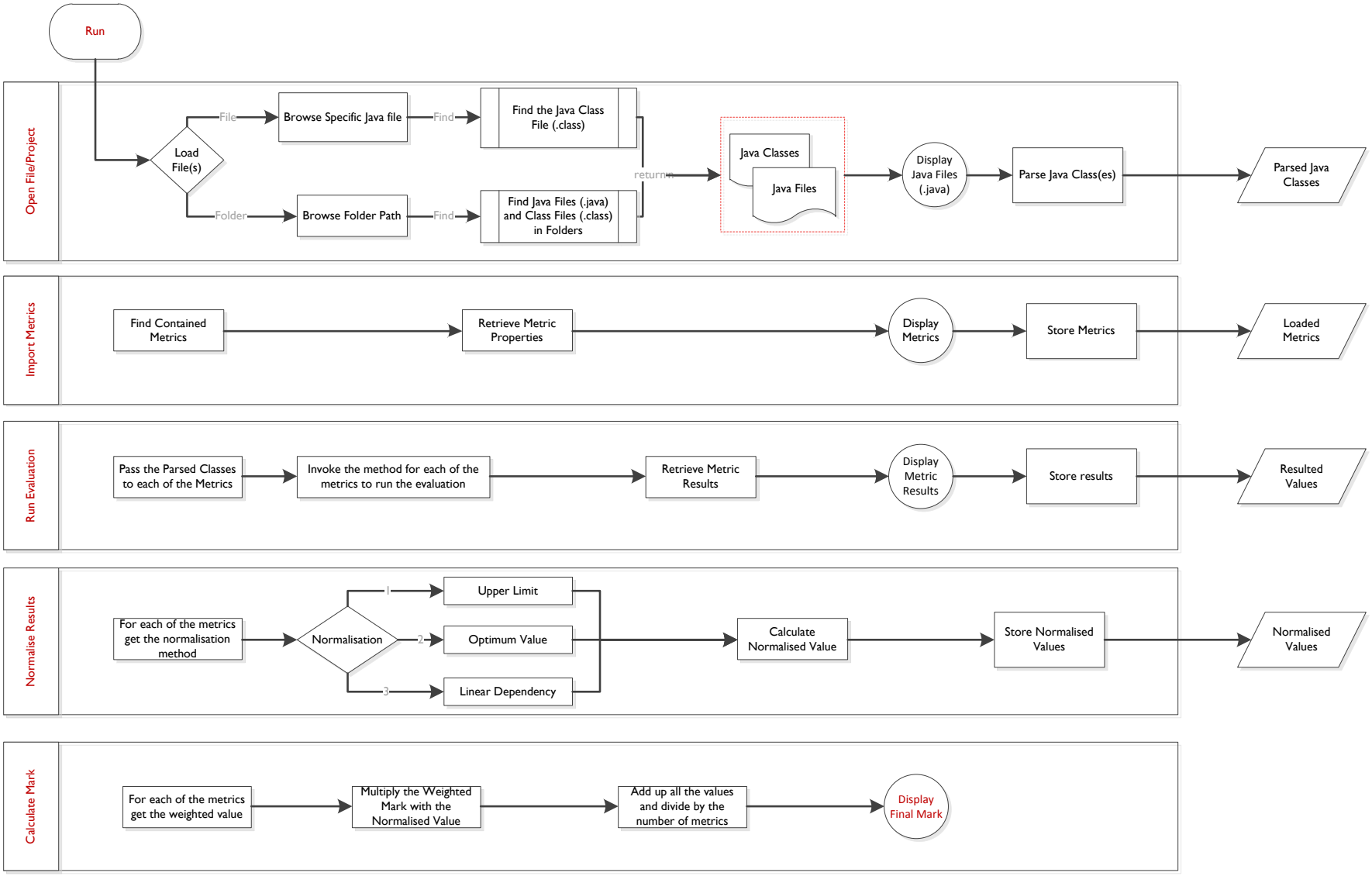


Bushmann et al. (1996)

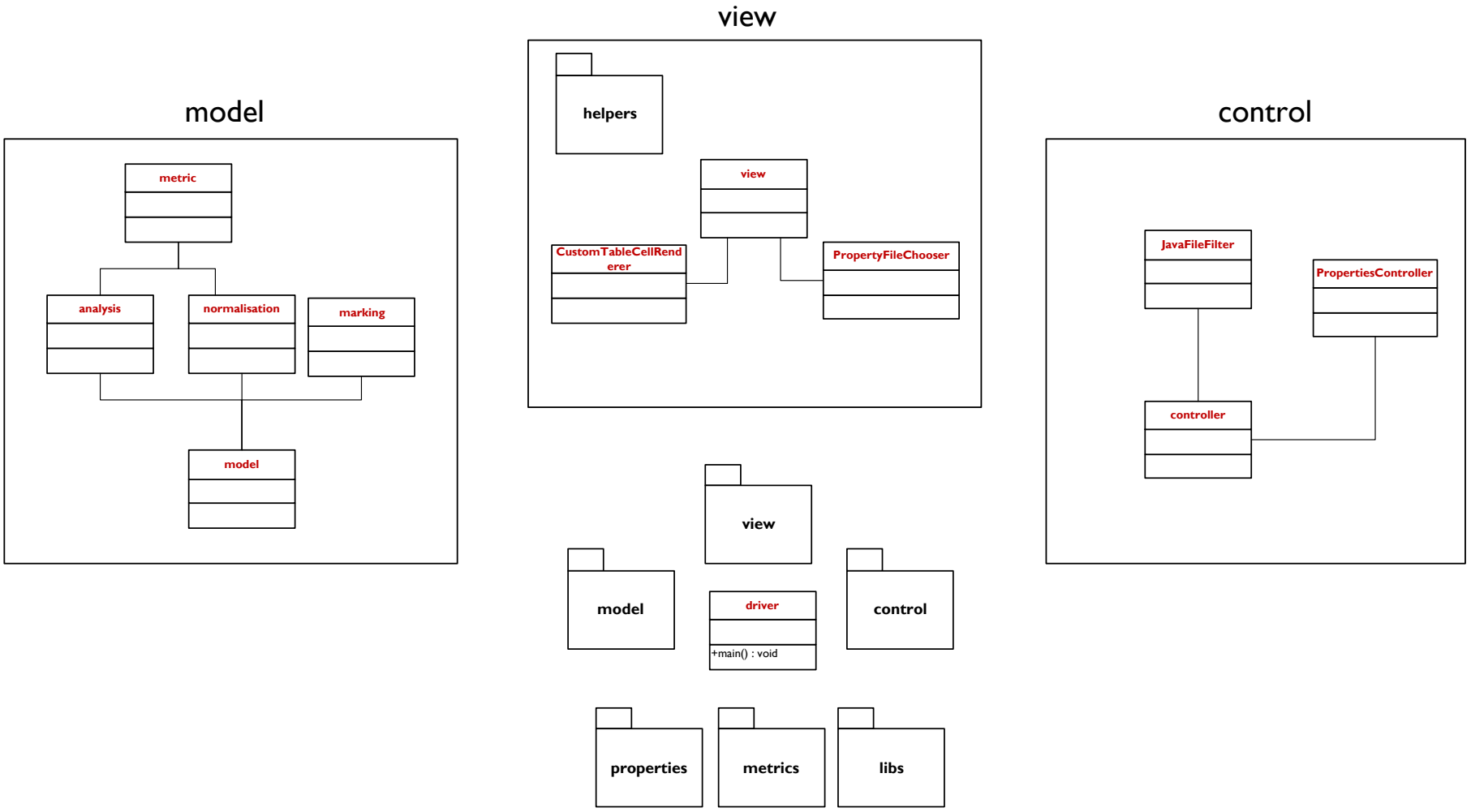
APPENDIX H: USE CASE DIAGRAM



APPENDIX I: FLOW CHART DIAGRAM



APPENDIX J: CLASS DIAGRAM

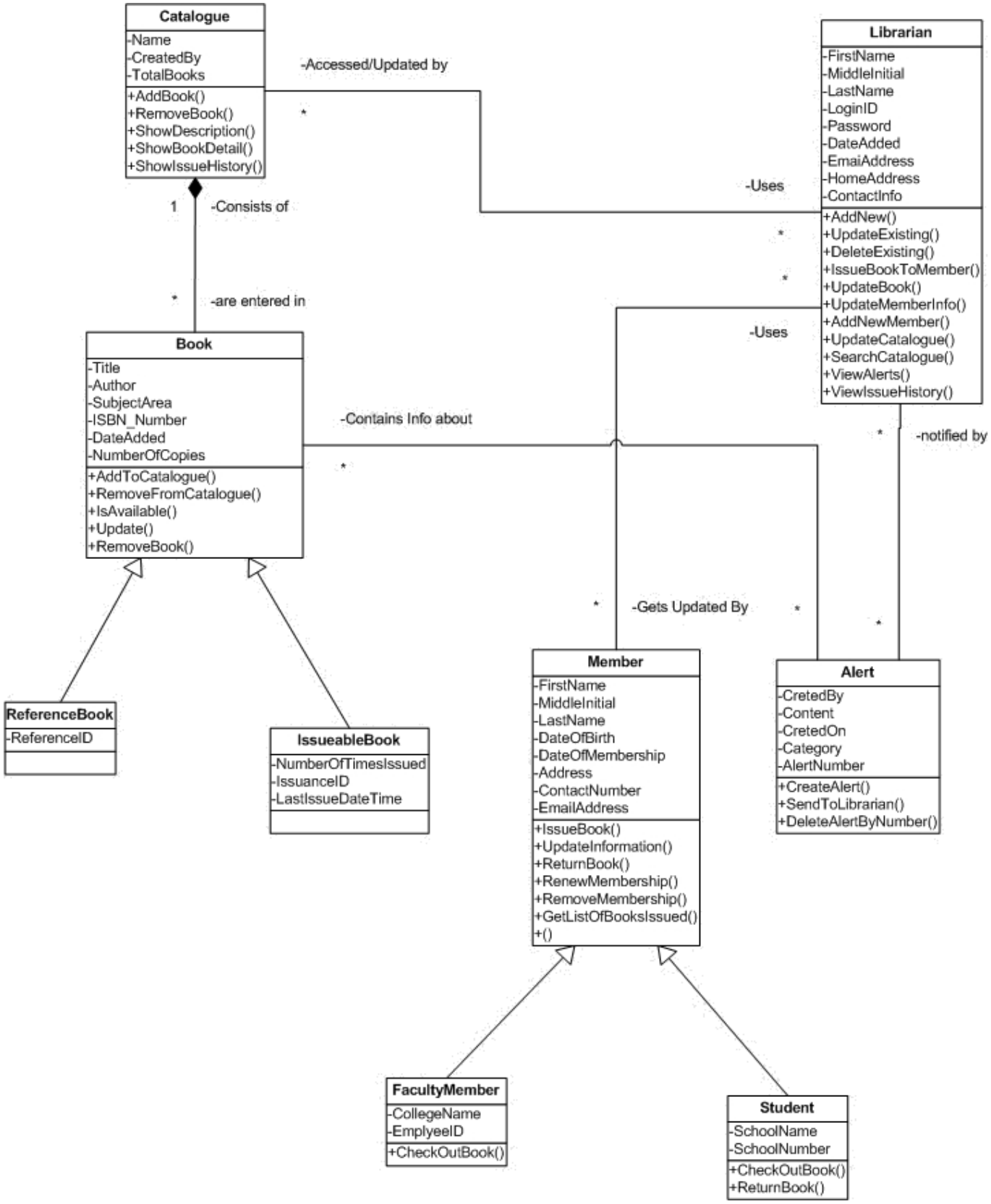


APPENDIX K: METRICS INTERFACE

```
package metrics;

/**
 * Metrics Interface
 *
 * @author Demetris Siampanias
 * @version (a version number or a date)
 */
import org.apache.bcel.classfile.*;
import java.io.*;

public interface metric
{
    public Boolean run();

    public int getNormalisationCode();

    public void initialise(File f);
    public String getMetricName();
    public String getMetricAcronym();
    public String getMetricDescription();
    public int getMeasuredValue();
    public int getMetricMinBoundary();
    public int getMetricMaxBoundary();
    public int getMetricWeight();
    public Boolean isMetricUsed();

    public void setNormalisationCode(int n);
    public void setMetricMinBoundary(int min);
    public void setMetricMaxBoundary(int max);
    public void setMetricWeight(int w);
    public void useMetric(Boolean u);
}
```

APPENDIX L: CLASS DIAGRAM FOR METRIC EXAMPLES

http://asadsiddiqi.wordpress.com/2008/02/21/object-oriented-design-a-class-diagram-walkthrough/



APPENDIX M: CCN TESTING CLASS

Java Class	Jasmin Disassembled Code

public class testing
{
    public testing()
    {
 	
    }
 	
    public void ifStatement(int y)
    {
        if (y >0)
        {
        }
        else if (y<0)
        {
        }
        else {}
    }
 	
    public void switchStatement(int y)
    {
        switch (y) {
            case 1: break;
            case 2: break;
            case 3: break;
            default:
System.out.println("Default");break;
        }
    }
 	
    public void trycatchStatement(int y)
    {
        try { }
        catch (Exception e) {}
        finally {}
    }
 	
    public void whileStatement(int y)
    {
        while (y>0)
        {
            y--;
        }
 	
    }
 	
    public void foreachStatement(int y)
    {
        for (int i=0; i<=y; i++)
        {
 	
        }
    }
}
	
	Method Name<init>
	   0: aload_0[42](1)
	   1: invokespecial[183](3) 1
	   4: return[177](1)
	
	Method NameifStatement
	   0: iload_1[27](1)
	   1: ifle[158](3) -> iload_1
	   4: goto[167](3) -> return
	   7: iload_1[27](1)
	   8: ifge[156](3) -> return
	  11: return[177](1)
	else: Label0
	else: Label1
	else: Label1
	
	Method NameswitchStatement
	   0: iload_1[27](1)
	   1: tableswitch[170](27) -> getstatic 2(1,
	goto[167](3) -> return = {27})(2, goto[167](3)
	-> return = {30})(3, goto[167](3) -> return =
	{33})
	  28: goto[167](3) -> return
	  31: goto[167](3) -> return
	  34: goto[167](3) -> return
	  37: getstatic[178](3) 2
	  40: ldc[18](2) 3
	  42: invokevirtual[182](3) 4
	  45: return[177](1)
	
	Method NametrycatchStatement
	   0: goto[167](3) -> return
	   3: astore_2[77](1)
	   4: aload_2[44](1)
	   5: athrow[191](1)
	   6: return[177](1)
	else: Label0
	
	Method NamewhileStatement
	   0: iload_1[27](1)
	   1: ifle[158](3) -> return
	   4: iinc[132](3) 1 -1
	   7: goto[167](3) -> iload_1
	  10: return[177](1)
	else: Label0
	else: Label1
	
	Method NameforeachStatement
	   0: iconst_0[3](1)
	   1: istore_2[61](1)
	   2: iload_2[28](1)
	   3: iload_1[27](1)
	   4: if_icmpgt[163](3) -> return
	   7: iinc[132](3) 2 1
	  10: goto[167](3) -> iload_2
	  13: return[177](1)
	else: Label0
	else: Label1
	ccn: 11

APPENDIX N: EMPTYVISITOR CLASS

CLASS EMPTYVISITOR

java.lang.Object
  └ **org.apache.bcel.classfile.EmptyVisitor**

---

public class **EmptyVisitor**
extends Object
implements Visitor

Constructor Summary

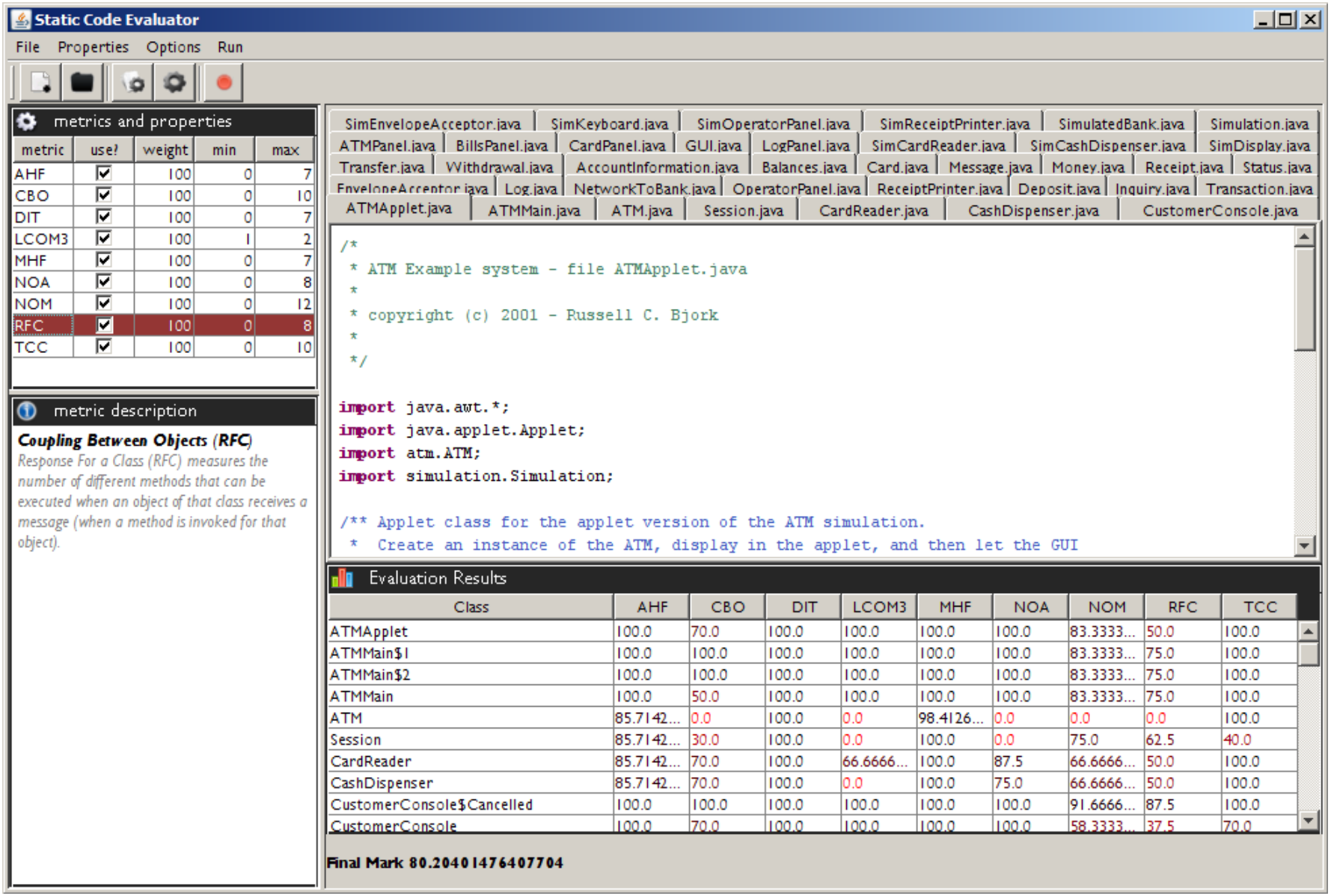
protected	**EmptyVisitor**()

Method Summary

void	**visitCode**(Code obj)
void	**visitCodeException**(CodeException obj)
void	**visitConstantClass**(ConstantClass obj)
void	**visitConstantDouble**(ConstantDouble obj)
void	**visitConstantFieldref**(ConstantFieldref obj)
void	**visitConstantFloat**(ConstantFloat obj)
void	**visitConstantInteger**(ConstantInteger obj)
void	**visitConstantInterfaceMethodref**(ConstantInterfaceMethodref obj)
void	**visitConstantLong**(ConstantLong obj)
void	**visitConstantMethodref**(ConstantMethodref obj)
void	**visitConstantNameAndType**(ConstantNameAndType obj)
void	**visitConstantPool**(ConstantPool obj)
void	**visitConstantString**(ConstantString obj)
void	**visitConstantUtf8**(ConstantUtf8 obj)
void	**visitConstantValue**(ConstantValue obj)
void	**visitDeprecated**(Deprecated obj)
void	**visitExceptionTable**(ExceptionTable obj)

void	**visitField**(Field obj)
void	**visitInnerClass**(InnerClass obj)
void	**visitInnerClasses**(InnerClasses obj)
void	**visitJavaClass**(JavaClass obj)
void	**visitLineNumber**(LineNumber obj)
void	**visitLineNumberTable**(LineNumberTable obj)
void	**visitLocalVariable**(LocalVariable obj)
void	**visitLocalVariableTable**(LocalVariableTable obj)
void	**visitMethod**(Method obj)
void	**visitSignature**(Signature obj)
void	**visitSourceFile**(SourceFile obj)
void	**visitStackMap**(StackMap obj)
void	**visitStackMapEntry**(StackMapEntry obj)
void	**visitSynthetic**(Synthetic obj)
void	**visitUnknown**(Unknown obj)

APPENDIX O: FINAL INTERFACE DESIGN



APPENDIX P: ATM METRIC RESULTS

CLASS	AHF	CBO	DIT	LCOM3	MHF	NOA	NOM	RFC	TCC
ATMApplet	100	70	100	100	100	100	83	50	80
ATMMain$1	100	100	100	100	100	100	83	75	100
ATMMain$2	100	100	100	100	100	100	83	75	100
ATMMain	100	50	100	100	100	100	83	75	50
ATM	85.7	0	100	0	98.4	0	0	0	100
Session	85.7	30	100	0	100	0	75	62.5	0
CardReader	85.7	70	100	66.7	100	88	66	50	100
CashDispenser	85.7	70	100	0	100	75	66	50	100
CustomerConsole$Cancelled	100	100	100	100	100	100	91	87.5	100
CustomerConsole	100	70	100	100	100	100	58.3	37.5	70
EnvelopeAcceptor	85.7	70	100	100	100	88	83.3	75	100
Log	100	60	100	100	100	100	58	37.5	100
NetworkToBank	85.7	50	100	100	100	75	66	50	100
OperatorPanel	85.7	70	100	0	100	88	83.3	75	100
ReceiptPrinter	100	80	100	100	100	100	83	75	90
Deposit$1	100	20	100	100	100	88	91	87.5	100
Deposit	85.7	0	100	100	100	75	58	37.5	100
Inquiry$1	100	30	100	100	100	88	91	87.5	100
Inquiry	85.7	0	100	100	100	88	66	50	100
Transaction$CardRetained	100	100	100	100	100	100	91	87.5	100
Transaction	92	0	100	0	100	0	33	0	0
Transfer$1	100	20	100	100	100	88	91	87.5	100
Transfer	85.7	0	100	100	100	63	50	25	100
Withdrawal$1	100	20	100	100	100	88	91	87.5	100
Withdrawal	85.7	0	100	100	100	75	58	37.5	80
AccountInformation	100	100	100	100	100	75	83	75	100
Balances	85.7	90	100	100	100	75	66	50	100
Card	85.7	100	100	100	100	88	83	75	100
Message	91.7	80	100	0	100	0	16	0	0
Money	85.7	100	100	0	100	88	41	37.5	100
Receipt$1	90.5	90	100	0	100	63	75	62.5	10
Receipt	90.5	40	100	75	100	63	66	50	100
Status	100	100	100	100	100	100	58	37.5	90
ATMPanel$1	100	80	100	50	100	75	83	75	100
ATMPanel	87	10	100	100	100	0	91	87.5	50
BillsPanel$1	100	90	100	0	100	88	83	75	90
BillsPanel	85.7	90	100	0	100	88	83	75	50

CardPanel$1	100	90	100	0	100	88	83	75	90
CardPanel	85.7	90	100	100	100	88	83	75	70
GUI	85.7	0	100	94.5	100	38	50	25	100
LogPanel$1	100	90	100	100	100	88	83	75	100
LogPanel$2	100	80	100	50	100	75	83	75	100
LogPanel	85.7	70	100	64.3	100	88	75	62.5	40
SimCardReader$1	100	80	100	50	100	75	83	75	100
SimCardReader	85.7	80	100	100	100	88	66	50	50
SimCashDispenser	85.7	90	100	80	100	88	83	75	90
SimDisplay	85.7	100	100	0	100	75	58	37.5	70
SimEnvelopeAcceptor$1	100	90	100	0	100	88	83	75	90
SimEnvelopeAcceptor	85.7	90	100	100	100	75	58	37.5	70
SimKeyboard$1	100	90	100	100	100	88	83	75	100
SimKeyboard$2	100	90	100	100	100	88	83	75	100
SimKeyboard$3	100	90	100	100	100	88	83	75	100
SimKeyboard$4	100	90	100	100	100	88	83	75	100
SimKeyboard$5	100	90	100	0	100	88	83	75	90
SimKeyboard	85.7	30	100	0	93.5	0	83	0	0
SimOperatorPanel$1	100	80	100	0	100	50	83	75	100
SimOperatorPanel$2	100	90	100	0	100	63	83	75	70
SimOperatorPanel	100	70	100	100	100	100	91	87.5	70
SimReceiptPrinter$1	100	90	100	0	100	88	83	75	100
SimReceiptPrinter	85.7	90	100	80	100	75	66	50	100
SimulatedBank$1	100	100	100	100	100	100	100	100	90
SimulatedBank$Failure	85.7	90	100	100	100	88	66	50	100
SimulatedBank$InvalidPIN	100	90	100	100	100	100	83	75	100
SimulatedBank$Success	100	80	100	100	97.1	100	58	50	100
SimulatedBank	85.7	10	100	0	91.1	25	33	0	0
Simulation	88.8	0	100	0	100	0	0	0	100

APPENDIX Q: USE CASE BASED TESTING

		Test Case **Tests**		Test Case **Conditions**		Test Case **Results**
Use Case						
load java class/project		Test that when a new class is loaded it is actually added in the List containing the classes. Loading a project is different because the tool automatically searches to find the classes; therefore we have to ensure that all the classes contained in a folder are identified successfully. Finally the classes loaded should be displayed on the tab.		the java class selected must be compiled/ the folder selected must contain compiled java classes		✓
change properties		Changing the weight, min, max boundary or choosing whether to use it in the evaluation must be applied to the metrics instantly and employed during the evaluation. Therefore it should be tested that this applies		valid metrics* must exist (*implement the metric interface and are compiled)		✓
save properties		When the user chooses to save the current properties an input box should appear for entering a name for the file to be saved as. It needs to be tested that the properties have been stored in the file specified and all the properties are as specified.		valid name* for the filename has been entered (*valid names for files)		✓
load properties		Loading the properties from one of the already saved file means that both the tool's metrics have to be configured.		the properties file must be valid		✓
select metric(s)		Choosing which metrics to use is similar to changing the properties of the metrics. Therefore this will be tested during the previous case.		at least one metric must be selected & weights > 0		✓
run evaluation		When the user selects to run the evaluation the metrics should evaluate the classes loaded and present the results to the user.		at least one class must be loaded in the system		✓
view results		The results when the evaluation is run should be displayed for the user to view. For each of the classes the value metric for each metric should be displayed.		the evaluation has run		✓
add metric class		Adding a new class to the metrics folder should directly be loaded in the system on the next start.		The metric class implements the metric interface		✓

APPENDIX R: TERMS OF REFERENCE

GC0174 MSC COMPUTING PROJECT

TERMS OF REFERENCE

Investigation in the automatic assessment of software's design quality. Development of a tool to support the automatic assessment of code using Object-Oriented metrics.

by Demetris Siampanias

Supervised by *Dr Ian Bradley*
Reviewed by *Dr Emil Petkov*

BACKGROUND INFORMATION: what, why, how?

A. MOTIVATION

In the last decade the object-oriented paradigm has decisively influenced the world of software engineering. On the other hand, in spite of the large acceptance of this paradigm, the design principles and the intimate mechanisms of object-orientation are not clearly understand and realised, and this fact has an outcome the development of poorly designed, difficult to understand and complicated object-oriented systems. Another issue that arises is when manually assessing the quality of such object-oriented systems as it is often difficult to measure the design quality e.g. degree of code reuse, complexity, maintainability or flexibility, etc., because of the number and size of classes.

Having that said, it would be helpful if there was a way to automatically provide information and formative feedback to the users to evaluate the design of their software, in order for them to acknowledge and understand how well their system is designed. Software metrics have been widely acknowledged as a successful way for measuring design quality. Despite that, different projects have different levels of required design qualities in the code. Thus this project will considers the utilisation of object-oriented metrics and their employment in a tool which will allow the selection of the required design attributes and the level of analysis depth and will automatically assess the design quality of code and provide both overall and particular feedback of the systems design quality attributes.

B. DESCRIPTION OF STUDY

"*What is not measurable make measurable*"

This statement by Galileo denotes the importance in science for quantifying observations as a way to understand and control the underlying causes. The same applies to Computer science as well, thus *metrics* have been developed to act as instruments for measuring software quality.

B.1 METRICS

Ebert and Morschel (1997) define software metrics as measures of development processes and the resulting work products and state that software metrics play an important role in analysing and improving quality of software work products during their development. Metrics can be used for multiple purposes and have a different role in the different situations. Such include their application for analysing source code as an indicator of quality attributes, providing programmers detailed feedback about their program's design quality and also as guidelines for when and where re-factoring is essential. Furthermore metrics have also been used as cost prediction, scheduling activities, and also as a productivity measurement (Clúa and Feldgen, 2008). Emam (2001) states that object-oriented metrics can also be used for quality estimation - as noted above - but also for risk management as well. The basic premise behind the development of object-oriented metrics is that they can serve as early predictors of classes that contain faults or that are costly to maintain (Benlarbi et al., 1999). Apparently utilisation of Object-Oriented metrics can be used for a variety of purposes and be applicable to assist the automatic assessment and evaluation of design.

Brooks and Buell (1994) justify that software metrics can be gathered at many levels of granullarity, but mainly at two particular levels; the class-level and the system-level. Class-level metrics measrure the complexity of individual classes contained in the system. System-level metrics on the other hand deal with collection of classes that comprise an object-oriented system. The software that will be developed will be aimed to cover and analyse the system classes and the system as a whole. As an example, evaluating the coupling level of the system all the classes will be taken into account as coupling measures the dependency of the class on each one of the other classes by automatically looking into the code and identifying the dependencies.

B.2 IDENTIFICATION OF METRICS

A considerable number of object-oriented metrics have been developed by the research community over the past few decades (Abreu and Carapuca, 1994; Benlarbi and Melo, 1999; Briand et al., 1997; Cartwright and Shepperd, 2000; Chidamber and Kemerer, 1994; Li and Henry, 1993; Tang et al., 1999) of which a few of these metrics have undergone some form of empirical validation, and some are actually being used by organisations as part of an effort to manage quality (Emam, 2001) . The basic premise behind metrics is that they capture some elements of object-oriented software complexity.

Object Oriented Metrics have mainly five characteristics, localisation, encapsulation, information, inheritance and object abstraction (Jamali, 2006). Jamali (2006) signified nine classes of Object Oriented Metrics. In each of then an aspect of the software would be measured; Size, Complexity, Coupling, Sufficiency, Completeness, Cohesion, Primitiveness (Simplicity), Similarity and Volatility.

Chidamber and Kemerer (1994) provide a set of language independent and well established metrics based in sound theory which have been one of the most widely adopted and experimentally investigated (Basili et al., 1995) set of metrics. Chidamber and Kemerer (1994) set of metrics is listed below:

- *Depth of Inheritance Tree (DIT)*
- *Number of Children (NOC)*
- *Coupling between object classes (CBO)*
- *Response for a Class (RFC)*
- *Weighted methods per class (WMC)*

Xenos et al. (2000) survey categorises metrics as *Object-Oriented* which are 'pure' object–oriented metrics and *Traditional* metrics for structural programming that could also be applied to object–oriented programming. Xenos et al (2000), survey provides a comprehensive and extended list (90 metrics) of both traditional and Object-Oriented metrics. While many of these metrics are based on seemingly good ideas about what is important to measure in object-oriented software to capture its complexity. Wakil and

Fahmy (no date) proceeded and presented classifications available for categorising software metrics, organised them in a taxonomy depicted in Appendix A. For the tools development a variety of metrics will be explored and categorised into taxonomies in order to aid the validation process following.

B.3 VALIDATION, EVALUATION AND SELECTION OF METRICS

In this research we focus on *evaluating* sets of object-oriented design metrics developed which have been identified earlier. The metrics will be evaluated based on comprehensive empirical validation (Briand et al., 1998; Benlarbi et al., 1999) of the metrics and the creation of a framework that will facilitate the specification of measurements and conditions that are necessary when assessing students Java code.

The measurements and criteria for selecting appropriate metrics for the product will be based on desirable characteristics of software metrics. Such characteristics are called meta-metrics and described by Xenos et al (2000). Automation is one of the meta-metrics as the effort required to automate a metric varies. Moreover, some metrics are relatively easy to implement (e.g. **L**ines **O**f **C**ode (LOC)), some more difficult (e.g. Class **H**ierarchy **N**esting **L**evel (HNL)) and some could not be implemented due to their nature (e.g. Percent of **P**otential **M**ethod uses **O**verridden (PMO)). Therefore a set of meta-metrics will be specified which will identify necessary design characteristics for the code evaluation.

C. HYPOTHESIS

"The development of a tool that will utilise metrics for evaluating code could assess and provide accurate formative feedback which will reflect on the code's design quality."

RESEARCH AREAS:

- Object-Oriented Metrics
- Automatic and manual assessment
- Object-Oriented design and implementation

PROJECT AIMS:

A. "The key aim of this project is to identify, categorise and choose Object-Oriented Metrics that could be utilised for assessing the design quality of students programming assignments"

B. "To design, implement, test an automatic assessment tool that can use Object-Oriented metrics for evaluating Java code's design for specified design quality attributes which can be used to provide informative feedback to the user."

PROJECT OBJECTIVES:

i. To review the problems involved when manually assessing software's quality and how automatic assessment would resolve those.
ii. To review how using metrics could be used in assessing the software's quality
iii. To identify and review the metrics available for automated assessment
iv. To select criteria and evaluate the metrics identified
v. To design and implement a tool that will automatically assess Java classes based on the selection of required metrics. The tool will provide an overall mark which indicates the design quality of the class being assessed.
vi. To test the project's hypothesis
vii. To evaluate the project and the project's success

DISSERTATION OUTLINE: THE OVERALL STRUCTURE OF THE DISSERTATION.

1. Introduction

2. Literature Survey

2.1 Software Design Quality Assessment

2.1.1 Design quality
2.1.2 Manual assessment
2.1.3 Automatic assessment

2.2 Object-Oriented metrics

2.2.1 Utilisation of OO metrics
2.2.2 Test other existing tools
2.2.3 Identification of Metrics
2.2.4 Evaluation and Selection of Metrics

3. Practical Work

3.1 Requirements Specification
3.2 Design of Product
3.3 Implement Product

3.4 Test Product

4. Hypothesis Testing

5. Evaluation

5.1 Evaluation of Project

5.2 Evaluation of Process

5.3 Evaluation of Product

6. Conclusions and Recommendations

7. Suggestions for future Development

RELATIONSHIP TO THE COURSE: Subject areas and relationship to course

This project involves working in several areas not covered by this current course. It requires an in-depth investigation of Object-Oriented metrics and how these can be used and employed for assessing software's design quality. It also involves refining techniques learnt such as Java during programming and UML during the design phase.

Furthermore and more particularly, in relation to the MSc Computing course, this topic makes use of a variety of areas which have been taught throughout the year and during BSc Computer Science: the use and knowledge in Java programming will be paramount to the success of the software and systems analysis and design will be used in order to plan out the proposed software and design a product which will meet the specified requirements and for it to be fitted to its purpose. For the product's development important will be the experience gained from Object Oriented system development module for developing systems, managing their design successfully, using design patterns and performing thorough tests for ensuring the tool's build quality.

More generally through the course important project management and development skills have been gained for controlling and handling the projects and products development process successfully and to ensure that the time will be utilised efficiently.

RESOURCES / CONSTRAINTS: Special resources and limitations to the project

In the first weeks of the project, the time will be entirely dedicated to research and familiarisation and gaining an understanding of Object-Oriented metrics, and more in generally identifying all the research material needed. After that, the time will be spent in the design, implementation and testing of the product.

Finally, the last weeks will be dedicated writing the report and evaluating the project and the product success.

To perform all these activities, there will not be any specific requirements of particular software or hardware. For accessing the resources; journal articles and books, the University's library and access points provide all the necessary resources needed and availability to most of the Online Digital Libraries for accessing Journals. For the programming part, BlueJ will be used which is available for free and can be accessed at the University's computers as well.

The project intends to use the facilities as provided by the school and if additional is required it is the student's responsibility to provide it.

References/Bibliography: sources of information necessary for the objectives.

BOOKS

Lanza, M., Marinescu, R., and Ducasse, S. (2005) *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc.

JOURNALS

Amandeep, K., Satwinder, S. and Kahlon K.S. (2009) 'Evaluation and Metrication of Object Oriented System', *Proceedings of the International MultiConference of Engineers and Computer Scientists 2009,* I, pp. 1063-1068 *IMECS 2009* [Online]. Available at: http://www.iaeng.org/publication/IMECS2009/IMECS2009_pp1063-1068.pdf (Accessed: 11 March 2010).

Basili, R. V., Briand, L. and Melo, L.W. (1995) 'A Validation of Object-Oriented Design Metrics as Quality Indicators', *IEEE Transactions on Software Engineering,* 22 (10), pp. 751 - 761 *ACM* [Online]. Available at: http://portal.acm.org/citation.cfm?id=239308 (Accessed: 11 March 2010).

Benlarbi, S., and Melo, W. L. (1999a) 'Polymorphism measures for early risk prediction', *In Proceedings of the 21st international Conference on Software Engineering ICSE '99*, pp. 334-344 *ACM* [Online]. Available at: http://doi.acm.org/10.1145/302405.302652 (Accessed: 11 March 2010).

Benlarbi, S., Emam, E.K., and Goel, N. (1999b) 'Issues in Validating Object-Oriented Metrics for Early Risk Prediction', *FastAbstract ISSRE*, pp. 1-2 [Online]. Available at: http://www.cistel.com/free_expertise/publications/Cistel-1999-10.pdf (Accessed: 11 March 2010).

Briand, L. C., Daly, J. W., and Wuest, J. (1997) 'A Unified Framework for Cohesion Measurement', *In Proceedings of the 4th international Symposium on Software Metrics (November 05 - 07, 1997)*, p. 43 *IEEE Computer Society* [Online]. Available at: http://www.computer.org/portal/web/csdl/doi/10.1109/METRIC.1997.637164 (Accessed: 11 March 2010).

Briand, L. C., Wüst, J., Daly, J., and Porter, V (1998) 'A Comprehensive Empirical Validation of Design Measures for Object-Oriented Systems', *In Proceedings of the 5th international Symposium on Software Metrics (March 20 - 21, 1998)*, p. 246 *IEEE Computer Society* [Online]. Available at: http://portal.acm.org/citation.cfm?id=823920# (Accessed: 11 March 2010).

Brito e Abreu, F., and Carapuça, R. (1994) 'Candidate metrics for object-oriented software within a taxonomy framework', *J. Syst. Softw.,* 26 (1), pp. 87-96 [Online]. Available at: http://dx.doi.org/10.1016/0164-1212(94)90099-X (Accessed: 11 March 2010).

Brito, F. (1995) 'Talk on "Design Metrics for Object-Oriented Software Systems"', *ERCIM Workshop Proceedings,* 95 (W001)*ERCIM* [Online]. Available at: http://www.ercim.eu/publication/ws-proceedings/7th-EDRG/7th-EDRG-contents.html (Accessed: 11 March 2009).

Brooks, C., and Buell, C. (1994) 'A Tool For Automatically Gathering Object-Oriented Metrics', *Proceedings of the IEEE 1994 National Aerospace and Electronics Conference (NAECON 1994),* 2, pp. 835-838 [Online]. Available at: http://www.comp.nus.edu.sg/~bimlesh/oometrics/8/00332952.pdf (Accessed: 11 March 2010).

Bruntink, M., and Deursen, A. (2004) 'Predicting Class Testability using Object-Oriented Metrics', *In Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE international Workshop (September 15 - 16, 2004)*, pp. 136-145 *SCAM. IEEE Computer Society, Washington, DC* [Online]. Available at: http://dx.doi.org/10.1109/SCAM.2004.15 (Accessed: 11 March 2010).

Cartwright, M., and Shepperd, M. (2000) 'An Empirical Investigation of an Object-Oriented Software System', *IEEE Trans. Softw. Eng.,* 26 (8), pp. 786-796 [Online]. Available at: http://dx.doi.org/10.1109/32.879814 (Accessed: 11 March 2009).

Cau, A., Concas, G. and Marchesi, M. (2006) 'Extending OpenBRR with Automated Metrics to Measure Object Oriented Open Source Project Success', *Workshop on Evaluation Frameworks for Open Source Software (EFOSS) Como*, pp. 1-4 [Online]. Available at: http://www.openbrr.org/como-workshop/papers/CauConcasMarchesi_EFOSS06.pdf (Accessed: 11 March 2010).

Chidamber, R. S., Darcy, P.D., and Kemerer, F.C. (1998) 'Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis', *IEEE Transactions on Software Engineering,* 24 (8), pp. 629-639 [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=34E92EB5A38C15817E1EE5427A9311E1?doi=10.1.1.61.2217&rep=rep1&type=pdf (Accessed: 11 March 2010).

Chidamber, S. R., and Kemerer, C.F. (1994) 'A Metrics Suite for Object Oriented Design', *IEEE Trans. Software Eng.,* 20 (6), pp. 476-493 [Online]. Available at: http://dx.doi.org/10.1109/32.295895 (Accessed: 11 March 2010).

Churcher, N., and Irwin, W. (2003) 'Object Oriented Metrics: Precision Tools and Configurable Visualisations', [Online]. Available at: http://ir.canterbury.ac.nz//handle/10092/3028 (Accessed: 11 March 2010).

Ciupke, O. (1999) 'Automatic Detection of Design Problems in Object-Oriented Reengineering', *In Proceedings of the Technology of Object-Oriented Languages and Systems (August 01 - 05, 1999)*, p. 18 *IEEE Computer Society, Washington, DC* [Online]. Available at: http://portal.acm.org/citation.cfm?id=833062# (Accessed: 11 March 2010).

Clúa, O., and Feldgen, M. (2008) 'Work in Progress - Object Oriented Metrics and Programmers' Misconceptions', *38th ASEE/IEEE Frontiers in Education Conference*,

pp. F3H-19-F13H-20 *IEEE* [Online]. Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=04720377 (Accessed: 11 March 2010).

Deligiannis, I., Sfetsos, P., Stamelos, I., Angelis, L. Xatzigeorgiou, R., and Katsaros, P. (no date) 'Assessing the Modifiability of two Object- Oriented Design Alternatives - A Controlled Experiment Replication ', [Online]. Available at: http://delab.csd.auth.gr/~katsaros/Experimental-Replication.pdf (Accessed: 11 March 2009).

Emam, E. K. (2001) 'A Primer on Object-Oriented Measurement', *Seventh International Software Metrics Symposium (METRICS'01)*, pp. 1-3 *IEEE* [Online]. Available at: http://www.computer.org/portal/web/csdl/doi/10.1109/METRIC.2001.915527 (Accessed: 11 March 2010).

Glasberg, D., El-Emam, K., Melo, W. and Madhavji, N. (2000) 'Validating Object-Oriented Design Metrics on a Commercial Java Application', *National Research Council 44146,* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.1103&rep=rep1&type=pdf (Accessed: 11 March 2010).

Guyomarc'h, J. Y., and Gueheneu, Y.G. (no date) 'On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics', [Online]. Available at: http://www.iro.umontreal.ca/~sahraouh/qaoose2005/paper4.pdf (Accessed: 11 March 2010).

Harisson, R., Counsell, S., and Nithi, R. (1997) 'An Overview of Object-Oriented Metrics', *In Proceedings of the 8th international Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97)*, p. 230 *IEEE Computer Society* [Online]. Available at: http://portal.acm.org/citation.cfm?id=831977# (Accessed: 11 March 2010).

Harrison, R., Counsell, S. J., and Nithi, R. V. (1998) 'An Evaluation of the MOOD Set of Object-Oriented Software Metrics', *IEEE Transactions on Software Engineering,* 24, pp. 491-496 [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.5269 (Accessed: 11 March 2010).

Jamali, S. M. (2006) 'Object Oriented Metrics. (A Survey Approach)', [Online]. Available at: http://ce.sharif.edu/~m_jamali/resources/ObjectOrientedMetrics.pdf (Accessed: 11 March 2010).

Kanmani, S., Sankaranarayanan, V., and Thambidurai, P. (2006) 'Evaluation of Object Oriented Metrics', [Online]. Available at: http://www.ieindia.org/pdf/86/pcn5fl4.pdf (Accessed: 11 March 2006).

Karunanithi, S., and Bieman, M.J. (1992) 'Candidate Reuse Metrics For Object Oriented and Ada Software', *Proc. IEEE International Software Metrics Symposium*, pp. 120-128 [Online]. Available at:

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.6703&rep=rep1&type=pdf (Accessed: 11 March 2010).

Khan, R. A., Mustafa, K. and Ahson, S.I. (2007) 'An Empirical Validation of Object Oriented Design Quality Metrics', *J. King Saud Univ.,* 19, pp. 1-16 *Comp. & Info. Sci.* [Online]. Available at: http://colleges.ksu.edu.sa/ComputerSciences/Documents/Paper%201-16.pdf (Accessed: 11 March 2010).

Koten, V. C., and Gray, A. (2005) 'An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability', *Information and Software Technology,* 48 (1), pp. 59-67 *Science Direct* [Online]. Available at: http://www.sciencedirect.com/ (Accessed: 11 March 2010).

Lanza, M., and Ducasse, S. (no date) 'Beyond Language Independent Object-Oriented Metrics: Model Independent Metrics', [Online]. Available at: http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Lan-Duc.pdf (Accessed: 11 March 2010).

Li, W., and Henry, S. (1993) 'Object-oriented metrics that predict maintainability', *J. Syst. Softw.,* 23 (2), pp. 111-122 [Online]. Available at: http://dx.doi.org/10.1016/0164-1212(93)90077-B (Accessed: 11 March 2010).

Marinescu, R. (1998) 'Using Object-Oriented Metrics for Automatic Design Flaws Detection in Large Scale Systems', *In Workshop Ion on Object-Oriented Technology (July 20 - 24, 1998) S. Demeyer and J. Bosch, Eds. Lecture Notes In Computer Science,* 1543, pp. 252-255 *Springer-Verlag, London,* [Online]. Available at: http://portal.acm.org/citation.cfm?id=704663# (Accessed: 11 March 2010).

Marinescu, R. (2005) 'Measurement and Quality in Object-Oriented Design', *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 1-4 *IEEE* [Online]. Available at: http://loose.upt.ro/download/papers/radum-phd-icsm05.pdf (Accessed: 11 March 2010).

Martin, R. (1995) 'OO Design Quality Metrics: An Analysis of Dependencies', *ROAD,* 2 (3) [Online]. Available at: http://www.objectmentor.com/resources/articles/oodmetrc.pdf (Accessed: 11 March 2010).

McCabe , J. T. (1976) 'A Complexity Measure', *IEEE Transactions on Software Engineering,* 2 (4), p. 308 [Online]. Available at: http://www.literateprogramming.com/mccabe.pdf (Accessed: 11 March 2010).

Pereira, S. R., and Csar, F.E. (2008) 'Metrics to Evaluate the Use of Object Oriented Frameworks', *Computer Journal,* 52 (3), pp. 288-304(217) *Oxford University Press* [Online]. Available at: http://www.ingentaconnect.com/content/oup/cj/2009/00000052/00000003/art00003 (Accessed: 11 March 2010).

Rosenberg, L. H. (1998) 'Applying and Interpreting Object Oriented Metrics', *Software Technology Conference,* [Online]. Available at: http://satc.gsfc.nasa.gov/metrics (Accessed: 11 March 2010).

Rosenberg, L. H., and Hyatt, L.E. (no date) 'Software Quality Metrics for Object-Oriented Environments', pp. 1-6 [Online]. Available at: http://www.cetin.net.cn/storage/cetin2/QRMS/ywxzqt24.htm (Accessed: 11 March 2009).

Salehie, M., Li, S., and Tahvildari, L. (2006) 'A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws', *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pp. 159 - 168 *IEEE* [Online]. Available at: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1631118 (Accessed: 11 March 2010).

Sarkar, S., Kak, C.A., and Rama, G.M. (2008) 'Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software', *IEEE Transactions on Software Engineering,* 24 (5), pp. 700-720 [Online]. Available at: http://dx.doi.org/10.1109/TSE.2008.43 (Accessed: 11 March 2010).

Schroeder, M. (1999) 'A Practical Guide to Object-Oriented Metrics', *IT Professional 1, 6 (Nov. 1999)*, pp. 30-36 [Online]. Available at: http://dx.doi.org/10.1109/6294.806902 (Accessed: 11 March 2010).

Sherif, J. S., and Sanderson, P. (1998) 'Metrics for object-oriented software projects', *J. Syst. Softw.,* 44 (2), pp. 147-154 *ACM* [Online]. Available at: http://dx.doi.org/10.1016/S0164-1212(98)10051-1 (Accessed: 11 March 2010).

Subramanian, G. a. C., W. (2001) 'An empirical study of certain object-oriented software metrics ', *Journal of Systems and Software,* 59 (1), pp. 57-63 [Online]. Available at: http://portal.acm.org/citation.cfm?id=507999.508003 (Accessed: 11 March 2010).

Subramanyam, R., and Krishnan, M.S. (2003) 'Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects', *IEEE Transactions on Software Engineering,* 29 (4), pp. 297-310 [Online]. Available at: http://portal.acm.org/citation.cfm?id=766682 (Accessed: 11 March 2010).

Talbi, T., Meyer, B. and Stapf, E. (2001) 'A metric framework for object-oriented development', *39th International Conference of Object-Oriented Languages and Systems (TOOLS 39),* (TOOL), pp. 164-172 [Online]. Available at: http://portal.acm.org/citation.cfm?id=884718 (Accessed: 11 March 2010).

Tang, M., Kao, M., and Chen, M. (1999) 'An Empirical Study on Object-Oriented Metrics', *Proceedings of the 6th international Symposium on Software Metrics*, p. 242 *IEEE Computer Society* [Online]. Available at: http://portal.acm.org/citation.cfm?id=520792.823979 (Accessed: 11 March 2010).

Tegarden, P. D., Sheetz, S.D. and Monarchi, E.D. (1996) 'Effectiveness of Traditional Software Metrics for Object-Oriented Systems', *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences,* 4, pp. 359-368 [Online].

Available at: http://www.acis.pamplin.vt.edu/faculty/tegarden/wrk-pap/HICSS25.PDF (Accessed: 10 March 2010).

Vaishnavi, V. K., Purao, S., and Liegle, J. (2006) 'Object-oriented product metrics: A generic framework', *Information Sciences,* 177, pp. 587–606 [Online]. Available at: http://iris.nyit.edu/~kkhoo/Spring2008/Topics/DS/OOProdMetrics_InfoSci-Vaishnavi-Purao-Liegle-2007.pdf (Accessed: 11 March 2010).

Wakil, E. M., and Fahmy, A. (no date) 'Software Metrics - A Taxonomy ', *Faculty of Computers and Information,* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.7086&rep=rep1&type=pdf (Accessed: 11 March 2010).

Wakil, E. M., Bastawisi, E.A., Boshra, M. and Fahmy, A. (no date) 'Object-Oriented Design Quality Models A Survey and Comparison ', *Citeseer* [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.59.1254&rep=rep1&type=pdf (Accessed: 11 March 2010).

Xenos, M., Stavrinoudis, D., Zikouli, K. and Christodoulakis, D. (2000) 'Object-Oriented Metrics - A Survey', *Proceedings of the FESMA 2000, Federation of European Software Measurement Associations, Madrid, Spain, 2000*, pp. 1-10 [Online]. Available at: http://citeseer.ist.psu.edu/528212.html (Accessed: 11 March 2010).

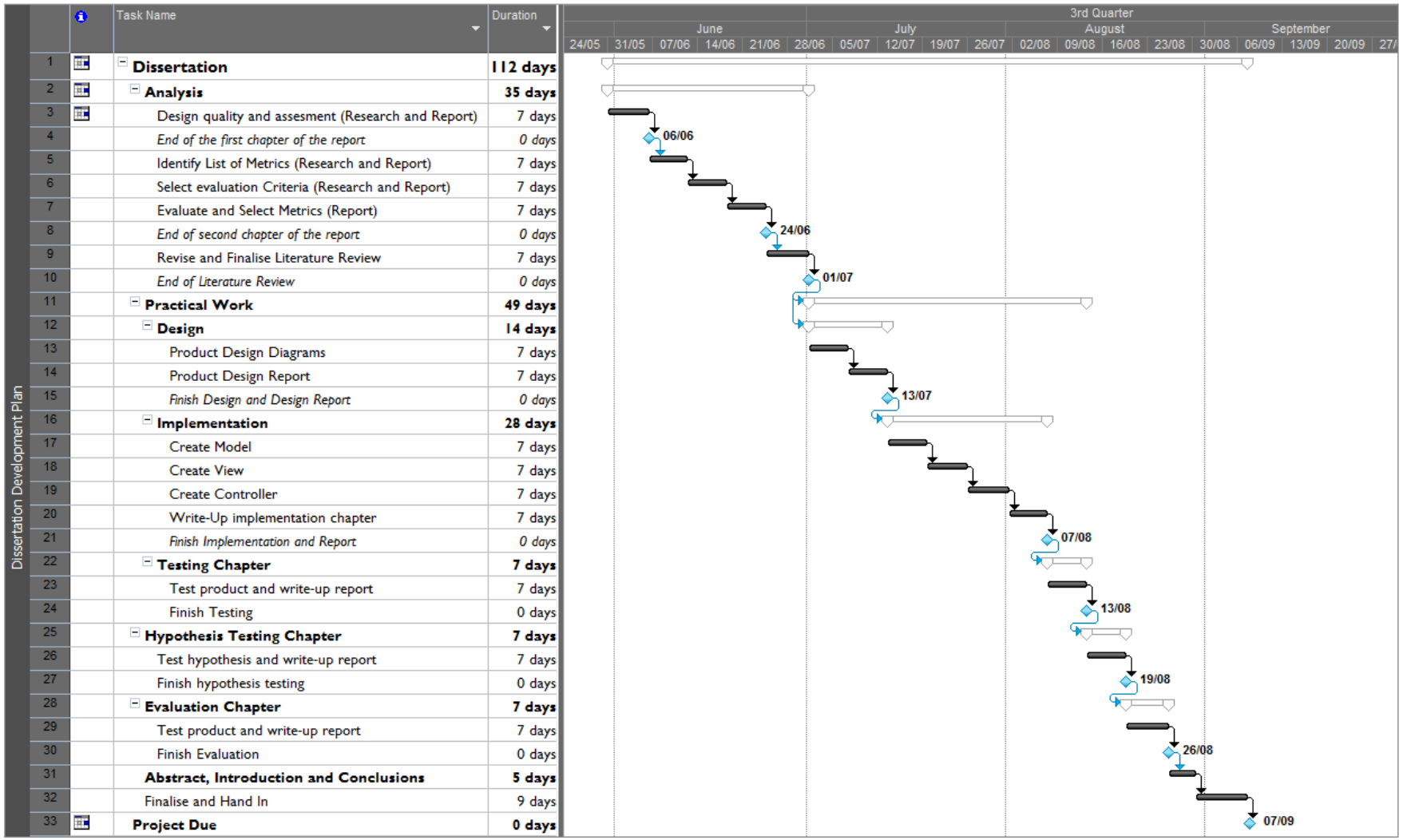
Xenos, M. a. C., D. (1997) 'Measuring Perceived Software Quality', *Information and Software Technology,* 39 (5), pp. 417-424 [Online]. Available at: http://dx.doi.org/10.1016/S0950-5849(96)01154-8 (Accessed: 11 March 2010).

Xie, T., Yuan, W., Mei, H., and Yang, F. (2000) 'JBOOMT: Jade Bird Object-Oriented Metrics Tool', [Online]. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.1260 (Accessed: 11 March 2010).

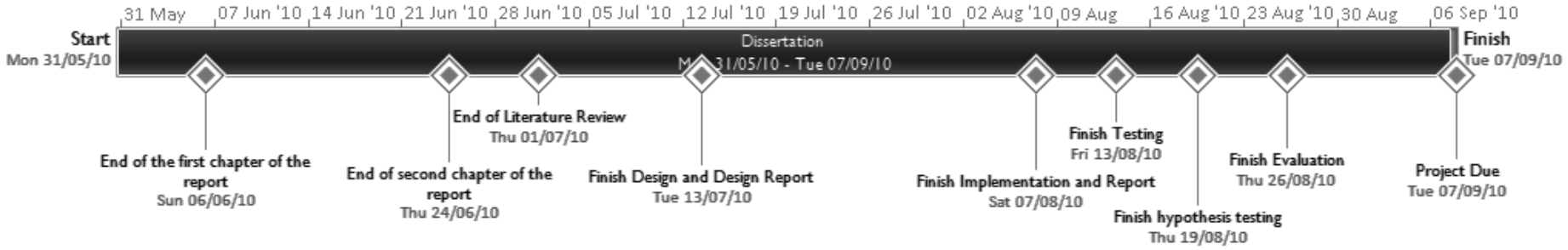
Xu, J., Ho, D. and Capretz, L.F. (2008) 'An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction', *Journal of Computer Science,* 4 (7), pp. 571-577 *2008 Science Publications* [Online]. Available at: http://www.scipub.org/fulltext/jcs/jcs47571-577.pdf (Accessed: 11 March 2010).

SCHEDULE OF ACTIVITIES: milestones, cross-referenced to objectives. Timeline and Gantt chart

GANTT CHART



PROJECT TIMELINE AND MILESTONES



APPENDIX A: Software Metrics Taxonomy (Wakil and Fahmy, no date)