6-1-2009

Software quality attribute measurement and analysis based on class diagram metrics

Dalia Rizk

The American University in Cairo

School of Science and Engineering

Software Quality Attribute Measurement and Analysis Based on Class Diagram Metrics

A Thesis Submitted to

Department of Computer Science and Engineering

in partial fulfillment of the requirements for

the degree of Master of Science

by Dalia Kamal Abd Alla Rizk

B.Sc. Computer Science, AUC, June 1999

under the supervision of

Prof. Dr. Hoda M. Hosny

April 2009

Dedication

I would like to dedicate this thesis to the soul of my late father Prof. Dr. Kamal Abd Alla Rizk, whom I wished so much that he would be around while presenting this research and during my graduation ceremony. I have to admit that if it were not for my father, I would not have considered going for the masters degree. I want to tell him that I miss him so much.

Acknowledgements

My first thanks and gratitude are for God who gave me the strength and persistence to finish this thesis.

My second appreciation goes directly to Prof. Dr. Hoda Hosny for initially her acceptance to be the supervisor of this research. Also, for her sincere devotion and guidance based on deep knowledge and patience through all the years that I have worked with her. Finally, for her keenness and encouragement to submit this thesis on time.

I thank Dr. Sherif El Kassas and Dr. Sherif Ali for their comments and their support during my research.

I thank Dr. Galal Hassan Galal Edeen, the Chairman of the Department of Information Systems, Faculty of Computers & Information – Cairo University, for his enlightening feedback about my research.

A special acknowledgement goes to Mr. Jürgen Wüst, the owner of SDMetrics Tool, for sending me the complete latest version of his tool and for giving me permission to use it in this research.

I would also like to thank Soha Makady (from Cairo University) for directing me to the SDMetrics Tool, and my friends Dalia Fakhry and Karim Wahba for their help and support and for being there whenever I needed either of them.

Last, but not least I would like to thank every member of my family, starting with my mother Prof. Dr. Samira Bishay for her encouragement throughout my academic studies and specially during the Masters Degree. Followed by my husband Eng. Ashraf Fouad and my son Andrew for bearing with me during the last phases of this research. I would also like to thank my parents-in-law for their motivation and encouragement.

ABSTRACT

University Name: The American University in Cairo

Thesis Title: **Software Quality Attribute Measurement and Analysis Based on Class Diagram Metrics**

By: Dalia Kamal Abd Alla Rizk

Supervisor: Prof. Dr. Hoda M. Hosny

Software quality measurement lies at the heart of the quality engineering process. Quality measurement for object-oriented artifacts has become the key for ensuring high quality software. Both researchers and practitioners are interested in measuring software product quality for improvement. It has recently become more important to consider the quality of products at the early phases, especially at the design level to ensure that the coding and testing would be conducted more quickly and accurately. The research work on measuring quality at the design level progressed in a number of steps. The first step was to discover the correct set of metrics to measure design elements at the design level. Chidamber and Kemerer (C&K) formulated the first suite of OO metrics. Other researchers extended on this suite and provided additional metrics. The next step was to collect these metrics by using software tools. A number of tools were developed to measure the different suites of metrics; some represent their measurements in the form of ordinary numbers, others represent them in 3D visual form. In recent years, researchers developed software quality models which went a bit further by computing quality attributes from collected design metrics.

In this research we extended on the software quality modelers' work by adding a quality attribute prioritization scheme and a design metric analysis layer. Our work is all focused on the class diagram, the most fundamental constituent in any object oriented design. Using earlier researchers' work, we extract a class diagram's metrics and compute its quality attributes. We then analyze the results and inform the user. We present our figures and observations in the form of an analysis report. Our target user could be a project manager or a software quality engineer or a developer who needs to improve the class

diagram's quality. We closely examine the design metrics that affect quality attributes. We pinpoint the weaknesses in the class diagram, based on these metrics, inform the user about the problems that emerged from these classes, and advice him/her as to how he/she can go about improving the overall design quality.

We consider the six basic quality attributes: "Reusability", "Functionality", "Understandability", "Flexibility", "Extendibility", and "Effectiveness" of the whole class diagram. We allow the user to set priorities on these quality attributes in a sequential manner based on his/her requirements. Using a geometric series, we calculate a weighted average value for the arranged list of quality attributes. This weighted average value indicates the overall quality of the product, the class diagram.

Our experimental work gave us much insight into the meanings and dependencies between design metrics and quality attributes. This helped us refine our analysis technique and give more concrete observations to the user.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Measurement lies at the heart of many systems that govern our lives [6]. Economic measurements determine price and pay increases. Measurements in radar systems enable us to detect an aircraft when direct vision is obscured. Medical system measurements enable doctors to diagnose specific illnesses. Measurements in atmospheric systems are the basis for weather prediction. Without measurement, technology cannot function [6]. But measurement is not the sole interest of professional technologists. Each of us uses it in our everyday life. Price acts as a measure of value of an item in a shop, and we calculate the total bill to make sure the shopkeeper gives us correct change. We use height and size measurements to ensure that our clothing will fit properly. When making a journey, we calculate the distance, choose our route, measure our speed, and predict when we will arrive at our destination (and perhaps when we need to refuel) [6].

The above examples present a picture of the variety of ways in which we use measurement. But there is a common thread running through each of the described activities: in every case, some aspect of a criterion is assigned a descriptor that allows us to compare it with others. In a shop, we can compare the price of one item with another. In the clothing store, we contrast sizes. And on our journey, we compare distance traveled to distance remaining. The rules for assignment and comparison are not explicit in the examples, but it is clear that we make our comparisons and calculations according to a well-defined set of rules. So measurement helps us to understand our world, interact with our surroundings and improve our lives [6].

Measuring quality is the key to developing high-class software [21]. In other words, assessing a software product helps with the improvement of software quality [34]. We need to measure quality in order to develop high-quality software where the safety and financial aspects are the main important aspects in our daily life [18]. In actual projects, quality metrics have been widely applied to manage software quality. This was mainly conducted by measuring the number of test items, the test coverage, and the number of faults in the test phase. This approach of relying much on testing is not satisfactory from a quality management viewpoint [19]. Therefore, it was thought to perform quality measurement on the coding level [21]. Then it was believed that assessing the quality of software at the design level would provide ease of use and higher accuracy for users [18].

Considering that software is getting larger and more complex, quality must be maintained from the early phases such as requirements analysis and design through coding [19]. Hence, the current trend in the software engineering field is to focus on the entire software development cycle rather than on the implementation part only [21].

1.1 Motivation

The degradation of software quality can incur significant costs on both the suppliers – who face dissatisfied customers, loss of market share, and rework of rejected systems – and the buyers, who receive faulty systems that fail to meet their mission goals [21].

Anselmo et al [1] borrowed from Tom DeMarco's book "Controlling Software Projects", the statement that "You can't control what you can't measure". They believe that before they can expect to improve productivity, they must measure it. Frakes et al [7] support this belief by stating that quality measurement is becoming an important factor in almost every company or organization. It is gaining more interest because it can assist both companies and researchers. In order for companies to improve productivity and quality, they must be able to measure their progress and identify the most effective quality measurement strategies [7]. While for the research community, traditional theory and methods about software quality have provided a foundation, yet further extensions are needed in order to cope with the new and more complex characteristics of software systems [18]. Anselmo et al [1] offered a framework that they believe is essential for making improvements in software productivity. They started by addressing issues concerning productivity of software development environments. There are no acceptable productivity benchmarks for a software environment [1]. Comparisons are generally based upon literature advocating a given method. Invariably they lack scientific data to support the claims. Software complexity grows rapidly when dealing with interactive user inputs, complex databases, dynamic graphics, networks, and so on. When functionality grows and software becomes more complex, development and support tools are put under the stress of a production environment. The more facilities contained in that environment to ease the development of these functions, the higher the productivity. The other 2 issues concerning productivity are scalability and reusability where the increasing complexity of software products stresses the scalability of the development environment in different

directions [1]. Therefore, many of the best software developers measure characteristics of the software to get some sense of whether the requirements are consistent and complete, whether the design is of high quality, and whether the code is ready to be tested [6]. Reuse is critical as a major justification for object-oriented programming (OOP) [1]. Unfortunately, there is no accepted definition of reuse nor a measure of its achievement. The main concern is that of measuring the effort required to reuse a software module in a new function. It is preferable to minimize the energy spent in development and support [1].

Anselmo et al [1] believe that before addressing measures for comparing software development environments, measures of the end product under development should be considered. Software systems are serving an ever-widening range of functionality. Poorly specified requirements are often cited as the cause for late and buggy software. Informed customers measure aspects of the final product to determine if it meets the requirements and whether it is of sufficient quality [6]. A more important factor appears to be the amount of functionality one must deal with [1]. It is required to quantify the size and complexity of the function space specified for a software product in order to determine the difficulty one faces in the development and support for that product. Effective project managers measure attributes of process and product to be able to tell when the software will be ready for delivery and whether the budget will be exceeded [6]. Another, product dimension that is required to be considered is the level of complexity of each function when measuring the difficulty in developing a piece of software [1]. As functionality and complexity grow, the number of opportunities for bugs multiplies and maintainers must be able to assess the current product to see what should be upgraded and improved [6]. Therefore, the quality of a software can be measured in terms of the availability of its specified functions; the time and cost to support that software and to maintain an acceptable level of availability, which must be determined by the users of that software [1].

Anselmo et al [1] present the properties of a software development environment that have been known to affect the man-hours and time to develop and support a software product. The first factor affecting productivity is independence whereby when attempting to reuse a module, one must be concerned with the independence of that module relative to its use

by different higher-level modules. The more a module shares data with other modules in a system, the higher is its connectivity to other parts of a system. The number of connections is measurable. The higher the connectivity, the lower the independence. Understanding the code is a major factor in software productivity, especially in the support phase of the life cycle of a product. Moreover, understandability of the architecture also contributes to the design of independent modules. Another factor affecting productivity is the flexibility whereby one can design a little, build a little, and test a little, thus growing a system incrementally to ensure components are meeting specifications and showing near-term results [1]. The third factor affecting productivity is visibility where the starting point is a visualization of the architecture on a modular basis and providing a one-to-one mapping into the detailed code. This can ensure design independence of modules while allowing visibility of the desired details. The last factor affecting productivity is abstraction where software should be broken into pieces such that the methods that produce them, including integration, can be examined experimentally [1].

Focusing on the entire software development life cycle gives software engineers the comprehensive knowledge they need in order to enhance software quality [21]. Such a broader focus supports early detection and resolution of quality problems, and the integration of product and process measurements lets engineers assess the interactions between them throughout the life cycle. It was found that quality metrics can be used to detect and remove problems with process and products in each phase [21]. Furthermore, it was found that, by using metrics throughout the life cycle, then in the test phase the progress of corrective action could be more quickly and accurately grasped [19]. Hence, engineers who are equipped with the knowledge to measure quality can better apply it to improve software quality throughout the development life cycle [21]. Since the field of software metrics is constantly changing and there is no standard set of metrics, and new measures are always being proposed; therefore, metrics extraction tools have to be updated frequently to handle these changes [28].

We were strongly motivated to contribute to the on-going research work on assessing software design elements based on quantitative measures.

1.2 Research Background

Bansiya et al [2] ascertained that the identification of a set of quality attributes that completely represent quality assessment is not a trivial task and depends upon many factors including management objectives, business goals, competition, economics, and time allocated for the development of the product. They proposed a new model (QMOOD) based on a class diagram, that has the lower-level design metrics well defined in terms of design characteristics, and quality is assessed as an aggregation of the model's individual high-level quality attributes [2]. According to Bansiya et al [2] the set of design quality attributes in QMOOD includes: "functionality", "effectiveness", "understandability", "extendibility", "reusability", and "flexibility". They selected specific existing metrics that could be calculated from class design information only, and they introduced five new metrics. We based our work in this research on their proposed metrics and design quality attributes.

More recent researches devised other models and approaches for measuring quality using different techniques. For example, Sharma et al [22] proposed a model for component based systems that can be used to estimate the quality of any component before using it in the final system. Wanger et al [33] presented a two-dimensional quality model approach that allows the structured elicitation and refinement of quality requirements using the activities of the stakeholders and their relationships with entities in the system that are documented in a quality model. Lamouchi et al [16] described a practical method that can be used to evaluate the expected quality of information systems. Stefan and Deissenboeck [32] suggested an integrated approach to quality modeling. Bhatti [4] stressed that measuring the quality of a system under construction is gaining higher interest especially when based on metrics collected from UML diagrams. Khan and Mustafa [12] proposed a model that addresses the low-level design metrics. Also, they use a set of empirically identified and weighted object-oriented design properties to assess testability.

Lakshminarayana et al [15] presented a new approach to aid understanding of object-oriented software through 3D visualization of software metrics that can be extracted from the design phase of software development. The focus of their work is on a metric extraction method and a new collection of glyphs for multi-dimensional metric visualization. Lakshminarayana et al [15] focus was on visually representing design

metrics to enhance their utility. They establish that visual representation can assist the software developers in quickly comprehending the values of the metrics and thereby aid in the detection of anomalies in the design. As a result, the design can be improved and, ideally, made more robust. The visual representation of the metrics created by Lakshminarayana et al's [15] tool for the classes of a UML class diagram was another important source of inspiration in this research work. They claim that values of the metrics for a class can be quickly obtained from their visual representation and conclusions about the class complexity can be drawn with ease [15].

1.3 Research Objective

The main objective of this research is to help project managers and software quality personnel assess the quality of a class diagram based on known design metrics and their own set of quality preferences.

Our approach in reaching the above objective involved a number of steps. We first identified the class diagram design metrics and their relationships with the most significant quality attributes from previous researches. Then we set thresholds on the metrics in order to build the assessment system. In order to give the user the capability of setting quality preferences we devised a priority scheme for the six quality attributes that we selected. Finally, we presented the class diagram assessment result to the user. Our result is not only comprised of a single score but of an analysis report in which we give the user feedback on the weaknesses in the diagram and wherever possible, on how they could be resolved.

The main ground for the research was the set of design metrics and quality attributes and the relationships between them. The Chidamber and Kemerer (C&K) suite of metrics, one of the first attempts at defining software metrics for object-oriented systems, was the main ground upon which previous researchers based their work and even added more of their own metrics. We based our work on the C&K suite and the researches that extended on it. We were also eager to verify their proposed metric computations and their relationships with quality attributes.

The idea of presenting a visual model was also very inspiring and we built an interactive tool to make our priority scheme and analysis reports visible to our target user.

1.4 Research Results

In this research we extended on the work of earlier researchers who set the class diagram design metrics and their relationships with software quality attributes. We specifically developed a class diagram assessment system, added a prioritization scheme for product quality attributes and an analysis reporting layer, and devised an interactive visual tool for the prioritization and analysis reporting.

We relied on some ready-made tools in drawing the class diagram and the collection of some of its metrics but we developed our own theoretical computations for the remaining metrics and for the quality attributes. The directions of the collected class diagram design metric values were very consistent with those prescribed by the researchers who developed the metric suites.

Within the class diagram's assessment system we suggested thresholds for most design metrics. According to the threshold comparison we present an analysis report that pinpoints the deficiencies in the class diagram and how these deficiencies may be resolved. We calculate quality attributes, based on the collected metrics, and we offer the user the option to set priorities for the quality attributes that they seek to satisfy for the software product under development. A weighted average value for their priority settings is given as an indicator of the quality of the diagram.

It was only through experimental testing that we could discover and analyze, the sometimes very complex relationships, between the various metrics and between the metrics and quality attributes and refine our analysis reporting system, accordingly.

We built an interactive tool that allows the user to set his/her quality attribute priorities and that visually represents the design metrics and the quality attribute values extracted from the class diagram. The tool calculates a weighted average for the quality attributes (based on the user's set of priorities), analyzes the metrics and presents a list of observations (based on thresholds) for each design metric. The observations guide the user in improving the class diagram and the improved diagram may be put through the tool again and the metrics recalculated. The new results would reflect the effect of the changes on the quality measurements and on the overall weighted average.

1.5 Document Outline

The remaining chapters in this document are organized as follows:

- Chapter two summarizes our literature survey on metric suites, automated tools, and the evaluation of metrics for Object-Oriented designs.
- Chapter three describes our solution approach and gives a brief explanation about our tool (SDAnalysis Tool).
- Chapter four presents our experimental tests on class diagram examples and the results obtained before and after enhancement based on our analysis reports.
- In Chapter five we discuss our experimental results and draw fine lines through our findings.
- Chapter six is the summary and conclusion chapter. It gives a summary of the research work and our contribution and discusses directions for further research that could extend on this research effort.

Chapter 2: Literature Survey

This chapter presents our literature survey findings about related topics. We first discuss some significant metric models and metric suites in section 2.1. Then we show how some researchers used automated tools to extract, record, manage, and visually represent the design metrics for a class diagram (section 2.2). In section 2.3 we illustrate how these researchers evaluated their models based on selected criteria and in section 2.4 we mention prioritization techniques applied in earlier work on software quality attributes.

2.1. Metric Models and Metric Suites

This section summarizes the relevant metrics and metric suites found in the literature. Metrics indicating the quality of Object-Oriented design are an example. Also, metrics implemented in the OOMet Tool [27] are being discussed. An overview of the MOOD metrics and the QMOOD is also presented.

2.1.1. Metrics indicating the quality of Object-Oriented Design

Liu et al [18] were concerned with the quality of an OO system design. They define a design to be a process that starts from a study of a domain problem and finally leads to some formal documentation. A software design is a model of the domain problem solution and it should capture and represent the user's requirements. It serves as a communication medium between the designer and the user on the one hand, and acts as a basis for implementation on the other hand. A design is a conceptual solution to a business problem while the software based on the design is just an implementation of the solution [18]. It is believed that system analysis and design must have a dominant position in the whole process of software development. According to Liu et al [18], Card and Grass, and Fenton have given a widely accepted and useful way to understand and evaluate the software quality, which they call a "factor-criteria-metrics" (FCM) model. As a first set, it is necessary to recognize the major factors that influence software quality. Secondly, some criteria need to be created for each factor. Finally, a set of metrics needs to be defined for each criteria (figure 2.1).

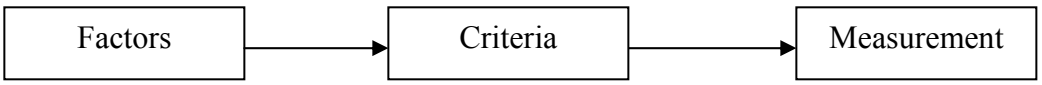


Figure 2.1: Software quality FCM model [18]

Hence, Liu et al [18] suggest that for each stage in the software lifecycle (requirements analysis, software design, implementation, testing, integration, and maintenance), a set of software quality factors should be identified in order to influence the quality of deliverables produced at each stage. Also, the corresponding criteria and metrics need to be created and defined for measuring and evaluating the quality of the products. The importance of software design is that it is concerned with accurately mapping the requirements from the analysis stage to the logical models for implementation. Liu et al [18] identify reliability, complexity, and reusability as the three major factors that influence the quality of object oriented software design. Reliability reflects the mapping between the requirements onto the design and the connection to its implementation. Complexity is the factor to be determined by the design method used and the personal experience of the designer. Reusability is the design quality, which leads to the reuse of software products that should be regarded of a better quality [18]. Furthermore, they identify a list of criteria for an OO design that indicate the quality of that design. A foremost important factor to judge the quality of a software design should be reliability, leading to the criteria of correctness and completeness [18]. Figure 2.2 illustrates the FCM model for software design quality.

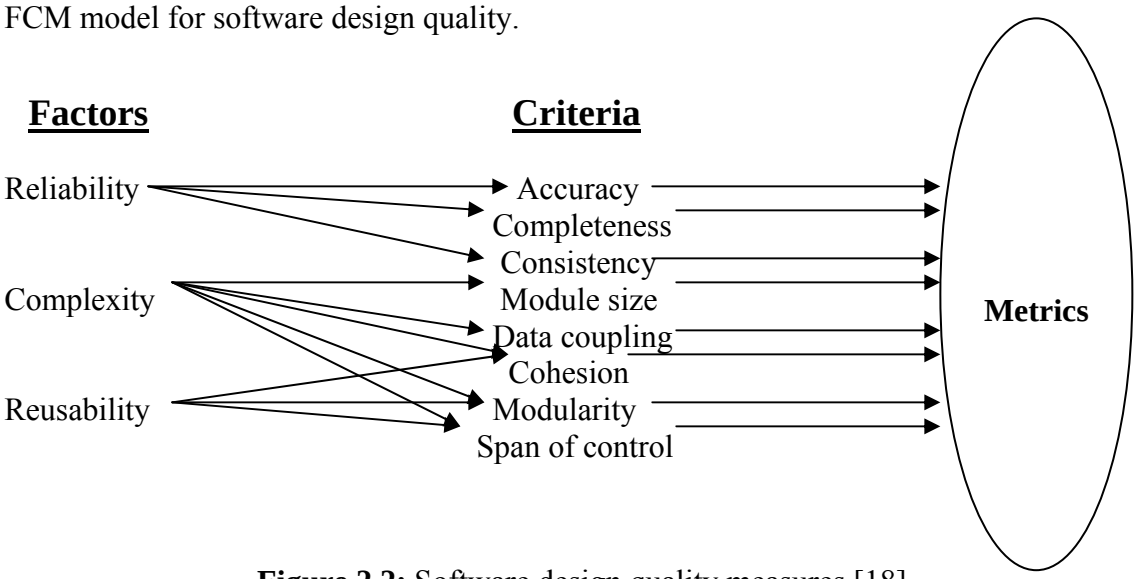


Figure 2.2: Software design quality measures [18]

This means a design should correctly capture and represent the user's requirements. A second important consideration of a quality design should be the readiness for implementation, which suggests the transformation from the design to an implementation and that should be rigid and straightforward. This will shorten the project lifecycle time and minimize the chance of incurring mistakes in implementation [18].

2.1.2. Metrics implemented in OOMet Tool

An implemented prototype of OOMetTool that provides an automatic support for metric data gathering was presented by Stiglic et al [27]. Although their final goal was to determine, identify and validate OO metrics that are suitable and significant for OO development, they started their investigation with two main objectives, namely: to compare styles of various C++ developers and to examine the extent of reuse. Their main interest was to use objective metrics, those that can easily be quantified, measured and automated [27]. Table 2.1 lists some of Chidamebr's metrics and some new metrics that were defined by the authors and implemented in the prototype of OOMetTool.

Class Level Metrics	**System Level Metrics**
Class level (DIT)	No. of files with source code
No. of Functions (WMC)	Lines of code (LOC)
Number of Children (NOC)	No. of classes (all, TOP, BOTTOM)
Response for a Class (RFC)	Avg. response
No. of parents	Avg. function in classes
% of public data members	Avg. depth of classes
% of protected data members	Avg. no. of children
% of private data members	% of abstract classes
% of public function members	No. of multiple inheritance
% of protected function members	% of non-member functions
% of private function members	% of TOP classes
No. of friends	% of BOTTOM classes

Table 2.1: Metrics implemented in OOMetTool [27]

According to Stiglic et al [27], during development of the prototype they obtained in-depth knowledge and understanding of all aspects of C++. Thus, they have already been developing a more suitable supporting tool that should help to find objective, cost effective and informative metrics with simple and precise definition.

The OOMetTool was intended for examination and analysis of OO projects developed in Borland C++. A MAP file is used since it contains a lot of information that are of interest to the project (e.g. list of classes, class members, idle functions) [27]. After extraction of useful information from the MAP file (function names, file names, classes, implemented classes, …) source code (H and CPP files containing declarations and definitions of classes/functions) are analyzed to obtain complete information on class structures and on hierarchical relationships between classes.

Stiglic et al [27] found that they don't have enough empirical data to make statistically valid assertions and therefore, they only presented qualitative interpretation of obtained results. Moreover, their results showed that multiple inheritance is rarely used. The average level of inheritance (1 to 2) of the newly developed classes indicates that OO developers involved in the research do not practice good design strategies which would lead to reusable components [27]. On the contrary, the rate of utilized reusable classes from libraries, mostly those related to user interfaces (e.g. OWL – Object Windows Library from Borland), is very high. This shows that developers have not yet adopted OO thinking. This is also confirmed by a great number of the so-called nonmember functions (functions not belonging to any class). Major violations of encapsulation have been identified for some developers. Violation of good design practice, where implementation is hidden from the user of an object, is strongly correlated to the developer's attendance at OO courses and/or the number of OO design methods, that a developer is familiar with [27]. An addition to their findings was that in the scope of OO approach a large amount of development effort has shifted from implementation to design. Also, design decisions greatly influence the quality attributes like reusability, maintainability and extensibility [27].

2.1.3. The Mood Metrics (MOOD)

Harrison et al [8] believe that analyzing object-oriented software in order to evaluate its quality is becoming increasingly important as the paradigm continues to increase in popularity. However, widespread adoption of object-oriented metrics in numerous application domains should only take place if the metrics can be shown to be theoretically valid, in the sense that they accurately measure the attributes of software which they were designed to measure, and have also been validated empirically. Therefore, Harrison et al [8] present a set of metrics for object-oriented design, called the MOOD metrics. They are being discussed from a measurement theory viewpoint, taking into account the recognized object-oriented features which they were intended to measure: encapsulation, inheritance, coupling, and polymorphism.

2.1.3.1. Theoretical Measurement Validation Issues

Harrison et al [8] based their investigation on the consideration of a number of criteria for a valid metrics set proposed by Kitchenham et al. According to Kitchenham et al [13], the main four theoretical measurement validation issues are:

1) For an attribute to be measurable, it must allow different entities to be distinguished from one another.
2) A valid measure must obey the Representation Condition, i.e., it must preserve all intuitive notions about the attribute and the way in which it distinguishes different entities.
3) Each unit of an attribute contributing to a valid measure is equivalent.
4) Different entities can have the same attribute value (within the limits of measurement error).

Harrison et al [8] further distinguish between direct measurement of an attribute which is measurement that does not depend on any other attribute, and indirect measurement which involves the measurement of one or more other attributes. They presented another distinction between internal attributes of a product or process (those attributes which can be measured purely in terms of the product itself), and external attributes of a product or process (those attributes which can only be measured with respect to how the product or process relates to entities in its environment).

Moving to Kitchenham et al [13], indirect measures calculated from a model must exhibit a number of properties:

1) Be based on a model concerned with the relationship among attributes as defined on specific abstract entities.
2) Be based on a dimensionally consistent model.
3) Exhibit no unexpected discontinuities.
4) Use units and scale types correctly.

2.1.3.2. Encapsulation

Harrison et al [8] discuss the merits of each of the six MOOD metrics from a theoretical validation viewpoint. They started by the encapsulation feature where they proposed the Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) metrics jointly. MHF is defined formally as:

$$\frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

where $M_d(C_i)$ is the number of methods declared in a class, and

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} \text{is\_visible}(M_{mi}, C_j)}{TC - 1}$$

where TC is the total number of classes, and

$$\text{is\_visible}(M_{mi}, C_j) = \begin{cases} 1 & \text{iff } j \neq i \wedge Cj \text{ may call } Mmi \\ 0 & \text{otherwise} \end{cases}$$

Thus, for all classes, C_1, C_2, ..., Cn, a method counts as 0 if it can be used by another class, and 1 if it cannot. The total for the system is divided by the total number of methods defined in the system, to give the percentage of hidden methods in the system [8]. AHF is defined in an analogous fashion, but using attributes rather than methods.

According to Harrison et al [8] some terms need to be defined. Data encapsulation is often taken to mean the power of a language to hide implementation details through (for example) the separate compilation of modules, the separation of interface from implementation, the use of opaque types, etc. Information hiding, on the other hand, can be defined in terms of the visibility of methods and/or attributes to other code. Information can be hidden without being encapsulated, and vice-versa.

For systems written in C++, the calculation of MHF is complicated by the existence of protected methods; this adjustment is problematic [8]. For a protected method in C++, the method is counted as a fraction between 0 and 1, calculated as[1]:

$$\frac{\text{number of classes not inheriting the method}}{\text{total number of classes} - 1}$$

2.1.3.3. Inheritance

Moving to the second object-oriented feature which is inheritance, Harrison et al [8] proposed the Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) metrics as:

$$\frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where

$$Ma(Ci) = M_d(Ci) + Mi(Ci)$$

and

M_d (Ci) = the number of methods declared in a class,

Ma (Ci) = the number of methods that can be invoked in association with Ci,

Mi (Ci) = the number of methods inherited (and not overridden) in Ci.

[1] The denominator has the value 1 subtracted from the total number of classes because the base class under consideration should not be included.

For MIF for each class C1, C2, …., Cn, a method counts as 0 if it has not been inherited and 1 if it has been inherited [8]. The total for the system is divided by the total number of methods, including any which have been inherited (i.e., methods which are inherited are counted as belonging to their base class as well as to all inheriting subclasses). AIF is defined in an analogous fashion. Thus, MIF and AIF measure directly the number of inherited methods and attributes respectively as a proportion of the total number of methods/attributes.

2.1.3.4. Coupling

The Coupling Factor (CF) metric was proposed as a measure of coupling between classes, excluding coupling due to inheritance [8]. CF is defined formally as:

$$\frac{\sum_{i=1}^{TC} \left\lfloor \sum_{j=1}^{TC} \text{is\_client}(C_i, C_j) \right\rfloor}{TC^2 - TC}$$

where

$$\text{is\_client}(C_c, C_s) = \begin{cases} 1 & \text{iff } C_c \Rightarrow C_s \wedge C_c \neq C_s \\ 0 & \text{otherwise} \end{cases}$$

and Cc => Cs represents the relationship between a client class, Cc, and a supplier class, Cs.

CF is calculated by considering all possible pair-wise sets of classes, and asking whether the classes in the pair are related, either by message passing or by semantic association links (reference by one class to an attribute or method of another class) [8].

2.1.3.5. Polymorphism

The Polymorphism Factor (PF) metric is the last object-oriented feature which Harrison et al [8] proposed as a measure of polymorphism potential. It is defined as:

$$\frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

where

$M_d (C_i) = M_n (C_i) + M_o (C_i)$

and

$M_n (C_i)$ = the number of new methods,

$M_o (C_i)$ = the number of overriding methods,

$DC (C_i)$ = the descendants count (the number of classes descending from C_i).

PF is the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations (the latter represents the case in which all new methods in a class are overridden in all its derived classes) [8]. Thus, PF is an indirect measure of the relative amount of dynamic binding in a system.

2.1.4. Project and Design Metrics

Kostecki [14] reviewed Lorenz et al's book and presented it by addressing the importance of metrics in software development. He believes that metrics have been used in the past to measure reliability and quality of the final software products, but now the emphasis is moving toward management of the software process as well as evaluation of intermediate software work products.

According to Kostecki, Lorenz et al presented at least nine projects written in Smalltalk, and two projects written in C++ [14]. The suite of metrics described in the book is divided into two main categories: Project Metrics and Design Metrics. Project metrics correspond with management issues, such as Application Size, Staffing Size, and Scheduling (of software deliverables). Design metrics are used to quantify the complexity, size and robustness of the object-oriented design being used.

Kostecki introduced the following set of seven attributes for design metrics that were given by Lorenz et al:

1) **Name:** A unique descriptive name for the metric.
2) **Meaning:** A description of the information which the metric gives to the user.
3) **Project Results:** Some graphical representations of the statistics collected were given.
4) **Affecting Factors:** The interdependency of metrics with other factors in the project is discussed. In some cases, metrics values will be affected by whether the

5) **Related Metrics:** The metrics proposed in this book are listed in related groups.
6) **Thresholds:** The authors use their experience with the metric to set some ranges for the values. The ranges not only delineate acceptable values, but also indicate undesirable ranges as well. This helps the users of the metrics by using this past experience to identify undesirable ranges as well. This helps the users of the metrics by using this past experience to identify anomalies in their own metrics. Kostecki [14] emphasizes that Lorenz et al point out that these anomalies may not be a problem, but their identification is a warning that some thought is necessary to assess the data.
7) **Suggested Actions:** According to Kostecki [14], the authors use their experience with the metric to help the readers understand what actions might be taken if the metric is either outside of the recommended threshold or is at an undesirable level.

2.1.4.1. Issues for Project Metrics

Kostecki [14] stated two of the three issues for project metrics being discussed by Lorenz et. al. to be as follows:

1) **Application Size** metrics are derived in order to provide management with an application specific comprehension of the amount of work needed. Lorenz et al have chosen measures which are focused on the design of the target application; for instance, Number of Key Classes, Number of Subsystems, and Number of Support Classes.
2) **Staffing Size** is also tied to the size and complexity of the application. Lorenz et al use two measures for this purpose: Person-Days Per Class and Classes Per Developer.

2.1.4.2. Measures for Design Metrics

According to Kotecki [14] there are 27 individual metrics defined in Lorenz et al book, but for the sake of brevity, he focused on the following:

1) **Method Size** consists of two measures: Number of Message Sends, and Lines of Code (LOC). The Number of Message Sends can be used to understand the intensity of communications within the application. Three types of messages:
    a) unary: where no arguments are passed.
    b) binary: where the message consists of one argument and is separated by a special selector.
    c) keyword: messages which contain one or more arguments.

    LOC remains a fairly straightforward way of expressing program size.
2) **Methods Internals** consists of two metrics: Method Complexity and String of Message Sends. Method Complexity attempts to replace such measures as McCabe's Complexity because of the shortfalls of the older, function-oriented measures in dealing with object-based systems [14].

2.1.5. Hierarchical Quality Model for Object-Oriented Design (QMOOD)

Bansiya et al [2] believe that due to the increase in demand for software quality, it has resulted in quality being more of a differentiator between products than it has ever been before. In a marketplace of highly competitive products, the importance of delivering quality is no longer an advantage, but a necessary factor for companies to be successful. Moreover, they think that the influence of an attribute may need to be changed by a weighting factor [2]. For large organizations with sophisticated networks and real-time processing, performance and reliability may be the most important attributes, whereas, for organizations that are in the multiplatform business, portability and extendibility are important attributes. As a result, Bansiya et al [2] highlight that the identification of a set of quality attributes that completely represents quality assessment is not a trivial task and depends upon many things including management objectives, business goals, competition, economics, and time allocated for the development of the product. Therefore, they presented a new model that has the lower-level design metrics well defined in terms of design characteristics, and quality is assessed as an aggregation of the model's individual high-level quality attributes [2].

2.1.5.1. Model Development

According to Bansiya et al [2] the methodology that they presented in the development of the hierarchical Quality Model for Object-Oriented Design (QMOOD) assessment extends Dromey's generic quality model that consists of three principal elements: product properties that influence quality, a set of high-level quality attributes, and a means of linking them. Bansiya et al [2] selected the ISO 9126 attributes – "functionality", "reliability", "efficiency", "usability", "maintainability", and "portability" – as the initial set of quality attributes in the QMOOD model. However, due to the obvious slant toward implementation rather than design, "reliability" and "usability" were excluded from the set. The term "portability" is more appropriate in the context of software implementation quality and was replaced with "extendibility" which better reflects this characteristic in designs [2]. Similarly, the term "efficiency" was replaced with "effectiveness" which better describes this quality for designs. The term "maintainability" also implies the existence of a software product and was replaced by "understandability" which concentrates more upon design characteristics. According to Bansiya et al [2] the set of design quality attributes in QMOOD includes: "functionality", "effectiveness", "understandability", "extendibility", "reusability", and "flexibility". Bansiya et al's [2] quality attributes' definitions are shown in table 2.2.

Quality Attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.
Understandability	The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
Effectiveness	This refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.

Table 2.2: Quality Attribute Definitions (adapted from [2])

2.1.5.2. Identifying Object-Oriented Design Properties

Bansiya et al [2] believe that the design properties of abstraction, encapsulation, coupling, cohesion, complexity and design size are frequently used as being representative of design quality characteristics in both structural as well as object-oriented development. Messaging, composition, inheritance, polymorphism, and class hierarchies represent new design concepts which were introduced by the object-oriented paradigm.

2.1.5.3. Identifying Object-Oriented Design Metrics

Each of the design properties identified in the QMOOD model represent an attribute or characteristic of a design that is sufficiently well defined to be objectively assessed by using one or more well-defined design metrics during the design phase. Bansiya et al [2] surveyed existing design metrics and suggested that there are several metrics that can be modified and used in the assessment of some design properties, such as abstraction, messaging, and inheritance. However, there are several other design properties, such as encapsulation, and composition, for which no object-oriented design metrics exist. Therefore, Bansiya et al [2] chose some existing metrics that could be calculated from design information only, and they also introduced five new metrics. Table 2.3 lists the complete suite of metrics used in QMOOD.

Metric	Name	Description
DSC	Design Size in Classes	This metric is a count of the total number of classes in the design.
NOH	Number of Hierarchies	This metric is a count of the number of class hierarchies in the design.
ANA	Average Number of Ancestors	This metric value signifies the average number of classes from which a class inherits information. It is computed by determining the number of classes along all paths from the "root" class(es) to all classes in an inheritance structure.
DAM	Data Access Metric	This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired. (Range 0 to 1)
DCC	Direct Class Coupling	This metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CAM	Cohesion Among Methods of Class	This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. A metric value close to 1.0 is preferred. (Range 0 to 1)
MOA	Measure of Aggregation	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations whose types are user defined classes.
MFA	Measure of Functional Abstraction	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOP	Number of Polymorphic Methods	This metric is a count of the methods that can exhibit polymorphic behavior. Such methods in C++ are marked as virtual.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class.
NOM	Number of Methods	This metric is a count of all the methods defined in a class.

Table 2.3: Design Metrics Descriptions (adapted from [2])

Furthermore, Basiya et al [2] identified the design components which are objects, classes, and the relationships between them. Another component that can be identified in object-oriented designs is class hierarchies that organize families of related classes. Thus, a set of components which can help analyze, represent and implement an object-oriented

design should include attributes, methods, objects (classes), relationships, and class hierarchies.

The diagram in Figure 2.3 illustrates the mapping of quality-carrying component properties to design properties. It also shows the assigning of design metrics to design properties. Finally, it presents the linking between design properties to quality attributes. Some of the design properties have positive influence on the quality attributes while on other quality attributes, they could have negative influence.

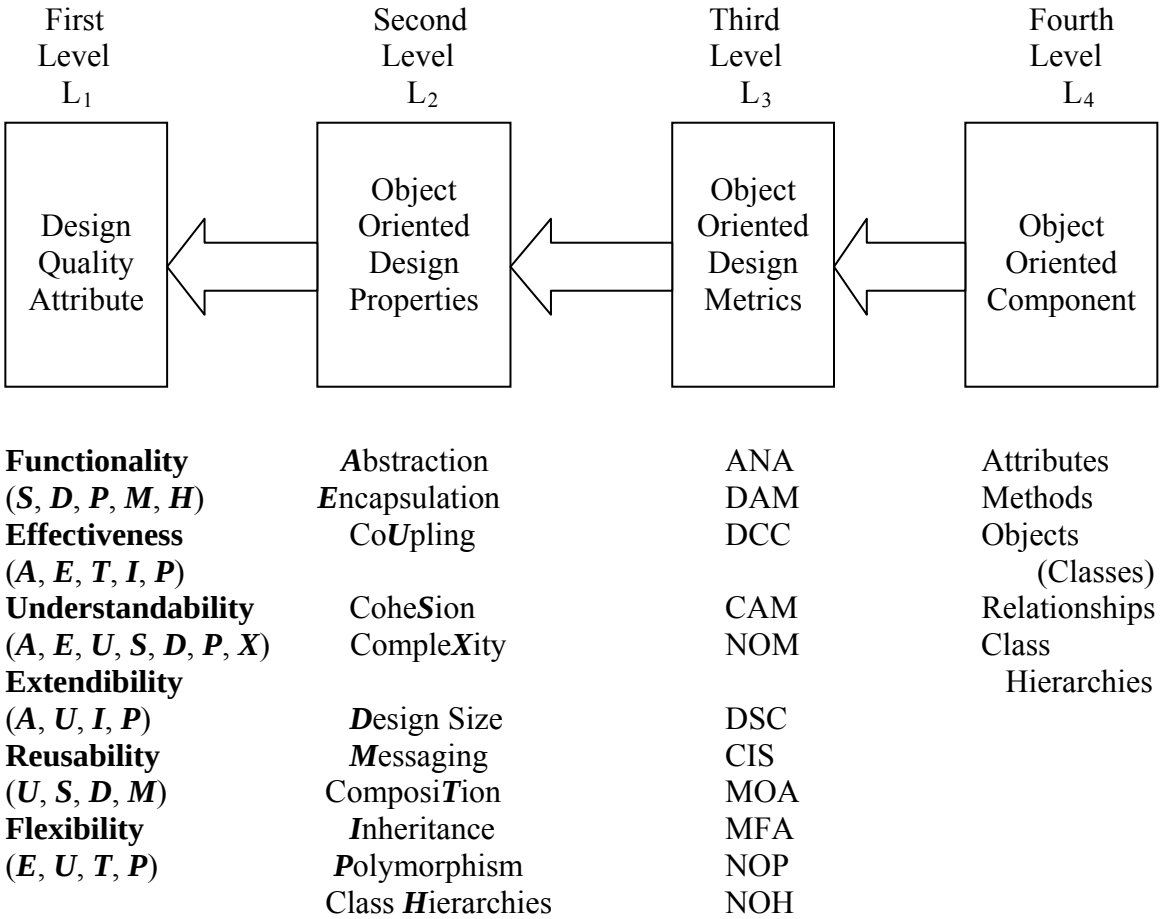


Figure 2.3: Levels and links in QMOOD (adapted from [2])

Bansiya et al [2] chose a scheme for weighing the influences on a quality attribute based on its simplicity and ease of application. The initial weighted values of design property influences on a quality attribute were then proportionally changed to ensure that the sum of the new weighted values of all design property influences on a quality attribute added

to ± 1, the selected range for the computed values of quality attribute. Table 2.4 shows the computation formulas for Quality Attributes as suggested by Bansiya et al [2].

Quality Attribute	Index Computation Equation
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation – 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation – 0.33 * Coupling + 0.33 * Cohesion – 0.33 * Polymorphism – 0.33* Complexity – 0.33 * Design size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction – 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

Table 2.4: Computation Formulas for Quality Attributes (adapted from [2])

2.1.6. Techniques for Collecting the Required Metrics

Wang's [34] research puts emphasis on the idea of assessing the software from its earliest stages where he uses software metrics as a measurement to conduct the assessment. When collecting software metrics from various components of a software product, he considers two important issues. The first issue is that only those metrics which interest us in measuring quality will be selected and he uses the ISO9126 model which is a quality model for product assessment to satisfy this issue. Secondly, he believes that a limited number of techniques for collecting metrics have been shown to be practical and then relates them to a quality characteristic [34].

2.1.6.1. ISO9126 Quality Model

The standard ISO9126 divides quality into six characteristics: functionality, reliability, usability, efficiency, maintainability, and portability. Functionality is defined as 'a set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.' Reliability is defined as 'a set of attributes that bear on the capability of software to maintain its level of

performance under stated conditions for a stated period of time.' Usability is defined as 'a set of attributes that bear on the effort needed for use and on the individual assessment of such use by a stated or implied set of users.' Efficiency can be decomposed into time behaviour: 'response time and processing time and on throughput rates', and resource behaviour: 'the amount of resources used and the duration of such use.' Maintainability requires analyzing the software to find the fault, making a change, ensuring that the change does not have side - effects, and then testing the new version. Portability is defined as 'a set of attributes that bear on the ability of the software to be transferred from one environment to another [34].

2.1.6.2. Metrics Collection Techniques

Wang [34] further classifies three basic metric collection techniques that could be used to assess any software product. Static analysis; where tools are used to measure the components without running them. This technique is often associated with the analysis of source code, but it can also be applied to specifications and designs in a formal or semi-formal notation. The second type is the execution analysis; where running the executable components is required. Manual inspection is the third type where the components are analyzed by hand. Each of these techniques is further broken into more specific types. There are four types of static analysis commonly used: anomaly checking, textual measurement, structural analysis, and test cross-referencing. The execution analysis has three techniques identified as: black-box testing, failure data collection, and test coverage. The commonly used approach in the manual inspection activity is by using checklists [34].

2.1.6.3. Types of Static Analysis

Anomaly checking, which is the first type in static analysis, applies only to formal language and specifically to source code. It is any undesirable feature of the code that may lead to a fault either during compilation, execution or porting to another environment [34]. It assesses the reliability feature as it identifies in the components those features that might be faults. It also can check for non-portable features of the source code leading to assessing the portability issue. There are two types of proposed metrics that might be

collected with information on anomalies that might be applicable to assess the reliability feature: first, simple counts of the anomalies (defined on the absolute scale), and secondly conformance to a language subset (on the nominal scale). Such a subset may be supported by a standard. For portability, any non-portable feature will be of interest such as: language extensions, non-standard library functions, calls to the OS, or embedded assembler code [34].

Textual measurement is another type of static analysis where it is based on the count of tokens or words in the document. The three main types of textual measurement are: measures of size such as lines of codes, or the density of comments [34]. Software size measures, based on the work of Halstead, are based on the source code and aim to predict the effort and difficulty associated with understanding the program. At the simplest level of counting the operands and operators of the program, these metrics can indicate size and amount of vocabulary. According to Wang [34], the third type of textual measurement is readability indices, defined on natural languages. It is also called the Fog index which is a measure of the readability of a passage of written text, the number of reported faults in a delivered software product and the number of person-days required to develop a system component [25]. Spelling checkers find faults in code comments [34]. Textual measurement can be used to determine the readability of documents in the case of natural language documents, and for source code it can be used to derive metrics which relate to maintainability (for program documentation) or usability (for user's documentation). Further illustration to quality characteristics is that an important part of the maintenance task understands how the existing software works. There are a number of proposed complexity metrics which measure the readability of the software. Among these factors are the size and structure of the modules. If the modules are too large then it becomes difficult to understand them. If on the other hand they are too small the maintainer will have to constantly switch attention between different parts of the code. Textual measure can be used to measure the size of the modules. In addition to maintainability, textual measurement affects the usability characteristic where the quantity and quality of the user manual will clearly have a bearing on the products' usability [34]. Textual metrics may be used to assess both the size of user documentation and its readability. Lines of code or text, number of characters, number of pages could be simple measures of size to be used

as metrics. The Halstead metrics are based on counts of the number of operands and operators in the code [34]. Operands are the variables, labels, and constants used in the program, and operators are the key words, arithmetic symbols, brackets, comparison operators, and other symbols (like ',' and ';'). Names of functions and procedures count as operands where the function is being defined, but as operators where it is being called. Comments and declarations are ignored. Thus the four base metrics can be defined as:

N1 The total number of operators

N2 The total number of operands

D1 The number of distinct operators

D2 The number of distinct operands

The size of the vocabulary V can be derived to be:

$$V = D1 + D2$$

and the length of the program:

$$N = N1 + N2.$$

The third technique related to static analysis is structural analysis which is applied to format notations such as source code or formal specifications [34]. Structural models such as flowgraphs or call graphs are derived from the code and from these, various structural metrics can be derived. In other words, structural analysis is based on deriving directed graph models of the software and then calculating metrics from these models. The most common models used are the control flow-graph which captures the algorithmic structure of a given module and the call graph which captures the interrelations between the modules in a compound module or subsystem. It is a graph with nodes represented by functions and their callers. The static analyzers for structural analysis typically have two parts: a front end and a back end. The front end reads in the source code, and outputs intermediate files. Front ends are always specific to a particular language. The back end reads in the intermediate files, calculates the metrics and displays them. Normally the graphs displayed by the back end can be used to create documentation or for inspection purpose [34]. We rather rely on metrics to assess the structural attributes than the graphs themselves because their layouts will depend heavily on the underlying implementation algorithm. Structural analysis can be used to decide on the maintainability where the structure of the various software components will affect how easy they are to understand

and test and this in turn will influence the maintainability of the software. There are already tools that structural metrics can be derived from. Logiscope in [34] defines directly from the flow-graph, the following metrics: number of nodes, number of edges, cyclomatic complexity number (defined as: edges – nodes + 2), and number of levels (the maximum number of nesting levels of control-flow constructs within the flow-graph). Regarding the call graph metrics, Logiscope calculates the hierarchical complexity which is the average number of modules on each level of the call graph. It also calculates the structural complexity which is the average number of calls per module. QUALMS in [34] is another tool that calculates other call graph metrics which are the maximum depth of call, number of recursions, Yin and Winchester metrics (a family of metrics which calculate how much the call graph deviates from a tree), and the re-use metrics (which determine to what extent modules are called by many different other modules).

The last static analysis technique presented by Wang [34] is the test cross-referencing where the test cases and functions of the software as described in the documentation can be cross-referenced to give measures of the functional coverage of the tests for gauging functionality. In simple words, test cross-referencing is a technique that connects functions of the product with specific test cases. The technique is based on following three steps. First is the extraction of the functionality from the functional specification describing the functional behaviors of the product. Then the specification of test cases with indication of the functions that are covered by each of them. Finally, the computation of functional coverage on the basis of tests and of their relationship with functionalities. The technique is oriented to the definition and monitoring of testing activities. The next logical stage is to perform the actual testing to determine whether the test cases pass or fail. Its key feature is that it creates a strong link between specifications and test documents (by means of the list of functions). It also gives a coverage measure which is closer to the user's perception than the actual testing coverage measure because the calculation of a measure for actual execution testing counts the hidden functions (supporting procedures) that may not appear on the specification. It is only possible to apply this technique to specification and test plans where they have been specially instrumented [34]. Therefore, test cross-referencing is related to the functionality quality characteristic which is mainly directed to the verification that all the functions are

expressed in the specifications of the product work correctly. TEFAX in [34] is a tool supporting the software testing and quality-control activities. It proposes some metrics that are related to assessment concerning test structure and functional coverage. Test-structure metrics are test redundancy (average number of tests covering functionality) and test power (average number of functionalities as stated in the documents covered by a test). Furthermore, there are two related metrics for actual execution testing namely test progress (percentage of tests actually executed) and functional coverage (percentage of passed functions, that is functions that were tested without causing a failure) [34].

2.1.7. The Use of an Intermediate Relation Set to Simplify Metrics Extraction

Succi et al [28] believe that the field of software metrics is constantly changing. There is no standard set of metrics, and new measures are always being proposed. Metrics researchers have to modify their existing parser tools in order to accommodate the new measures. Therefore, they presented a paper that details the use of an intermediate relation set to decouple code parsing from metrics analysis [28]. Parsers simply generate a set of intuitive relations, which a separate analyzer uses as input to compute arbitrary metrics. Then, new metrics simply have to be specified in terms of these relations. More specifically, the language parsing should be decoupled from the metrics analysis portion of the process. This requires an additional layer of abstraction with an associated intermediate representation [28].

This is done by presenting a high-level, metrics-oriented intermediate representation in the form of a set of relations. The relations describe the interaction between different language entities, such as classes and methods. Metrics can be calculated by directly querying the relation set. For example, a metrics researcher who wants to calculate the depth of inheritance tree for a class needs to look at the inheritance hierarchy to deduce the measure. The metrics researcher should not have to deal with language parsing production concepts such as declarations, class specifiers, and base clauses in order to calculate the measure [28].

2.1.7.1. Advantages and Disadvantages of Modular Metric Extraction

Succi et al [28] believe that the disadvantage of their approach lies in adding an extra layer of abstraction which leads to elongating the initial development time. However, the savings in maintenance effort later on in the development lifecycle offset this disadvantage. On the other hand, they believe that the advantages lie in that a user only needs to deal with the high-level, metrics-oriented intermediate representation when adding or modifying metrics to be calculated [28].

2.1.7.2. C++ Metrics Extractor

To test the overall concept of the relation set, Succi et al [28] built a tool to calculate OO design metrics from relations extracted from C++ source code. They presented only seven relation types in their relation set (shown in table 2.5 below). These have been chosen to specifically facilitate the calculation of certain OO design metrics.

Relation	**Description**	**Simple Example**
hasLOC(entity, *x*)	The specified entity has *x* lines of code.	hasLOC(Stack, 6)
hasMethod(entity, method)	The specified entity has the specified method.	hasMethod(Stack, Stack::push)
hasAttribute(entity, attribute, typename)	The specified entity has an attribute of the specified type.	hasAttribute(Stack, Stack::size, int)
Extends(entity, class)	The specified entity is a specialization of the specified	extends(Stack, Collection)
hasClass(entity, class)	The specified entity contains the specified (inner) class.	hasClass(Stack, Stack::Iterator)
calls(entity, method, *x*)	The specified entity called the specified method *x* times.	calls(Stack::push, Stack::isFull, 1)
UsesAttribute(entity, attribute, *x*)	The specified entity uses the specified attribute *x* times.	usesAttribute(Stack::isFull, Stack::size, 2)

Table 2.5: Relation types (adapted from [28])

2.1.7.3. Application of Relation Set

The first module of the tool takes preprocessed C++ source code as input, and writes the extracted relation set as output [28]. The second module takes the relation set as input and calculates the following for each class:

- LOC (Lines Of Code in terms of number of semicolons)
- WMC (Weighted Method Count)
- DIT (Depth of Inheritance Tree)
- NOC (Number Of Children)
- CBO (Coupling Between Object classes)
- RFC (Response For a Class)
- LCOM (Lack of Cohesion between Methods)

Succi et al [28] expressed the CK metrics in terms of the relations using a set-based notation. Table 2.6 shows an example of these metrics:

Metric	**Expressed in terms of relations**
WMC	$WMC(X) = \lvert\{i : hasMethod(X, i)\}\rvert$
DIT	$DIT(X) = \begin{cases} 1 + \max(\{DIT(i) : i \in I\}), & I \leftarrow \{i : extends(X, i)\} \wedge I \neq \emptyset \\ 0, & I = \emptyset \end{cases}$
NOC	$NOC(X) = \lvert\{i : hasClass(X, i)\}\rvert$

Table 2.6: Expression of CK metrics in terms of relations (adapted from [28])

As shown in table 2.6, the relations can be easily used to formally express metrics. The metric values can then be calculated directly using these expressions [28].

2.1.8. Semantic Metrics from Requirements or Design Specifications

Software metrics can provide an automated way for software practitioners to assess the quality of their software. The earlier in the software development life cycle this information is available, the more valuable it is, since changes are much more expensive to make later in the life cycle. Stein et al [26] presented a research that focuses on using

semantic metrics to assess systems that have not yet been implemented. They chose semantic metrics as they do not rely on the syntax or structure of code, they can be computed from requirements or design specifications before the system is implemented.

2.1.8.1. Background of Semantic Metrics

According to Stein et al [26] a suite of semantic metrics that is calculated based on concepts in a knowledge base that are associated with a class or method was introduced by Etzborn and Delugach. The suite includes the following metrics:

- LORM (logical relatedness of methods): the number of relations in a class divided by the number of pairs of methods in the class.
- CDC (class domain complexity): the sum of the concepts associated with a class, each multiplied by a weighting factor, plus their associated conceptual relations.
- RCDC (relative class domain complexity): the class's CDC value divided by the maximum CDC value for any class in the system.
- KCI (key class identity): 1 if the class's RCDC value is at least 0.75; 0 otherwise.
- COa (class overlap): the sum of the concepts in common between two classes, divided by the total number of unique concepts in either class, computed for all classes in the system and divided by the number of classes in the system [26].

2.1.8.2. Design Metrics

According to Stein et al [26], Bieman and Kang proposed a new way to assess the cohesion of a module (here, a procedure or function) from the design alone. They defined six types of relationships that could exist between any pair of outputs of a module. These relationships are:

- Coincidental Relationship (R1): the two outputs do not depend on each other, and they don't depend on any common input.
- Conditional Relationship (R2): the two outputs depend on the same input, and that input is a condition in a branch control structure.

- Iterative Relationship (R3): the two outputs depend on the same input, and that input is a condition in a repetition control structure.
- Communicational Relationship (R4): two outputs depend on the same input, and that input is not a condition in any branch or repetition control structure.
- Sequential Relationship (R5): one output depends on the other.
- Functional Relationship (R6): the module has only one output [26].

Some of these relationships might still be difficult to identify during the design phase, particularly the ones that depend on whether the input is a condition in a control structure; however, this metric is a step in the right direction for true design metrics [26].

Moreover, Bieman and Kang defined three other design level cohesion metrics [26]. They defined isolated components to be those that affect only one output of the module; essential components are those that affect or depend on all outputs of the module. In this context, a component is an input or output of a module. From these definitions, Bieman and Kang's metrics are defined as follows [26]:

- LC (loose cohesiveness): the number of isolated components divided by the number of components in the module.
- TC (tight cohesiveness): the number of essential components divided by the number of components in the module.
- MC (module cohesiveness): the sum over all components of the connectedness of each component, divided by the number of components in the module [26].

Stein et al [26] presented another study of design level metrics that was performed by Bansiya and Davis that defined a model called QMOOD, containing four levels to be analyzed in object-oriented design:

- Components: objects, classes, and relationships.
- Metrics (several new ones)
- Properties: abstraction, encapsulation, coupling, cohesion, complexity, and size.
- Quality attributes: functionality, effectiveness, understandability, extensibility, reusability, and flexibility [24].

2.1.8.3. Metrics Used to Predict Aspects of Software Quality

Stein et al [26] explained the studies done by Basili, Briand, and Melo and Briand et al that dealt with analyzing existing metrics for their use as predictors of probability of fault. Probability of fault is the likelihood that a fault will be detected in a module during an inspection. Basili, Briand, and Melo found that fault probability had significant positive correlation to DIT (depth of inheritance tree), RFC (response for class), and CBO (coupling between objects). They also found a significant negative correlation to NOC (number of children), which they attributed to more design and testing effort being expended on classes on which other classes will be based [26]. Moreover, Briand et al began with a set of 49 metrics compiled from 12 different sources. Using logistic regression, they found that the best model contained 11 of the original metrics. This model found 95% of the faults in the system, and 85% of the modules it flagged as probably having faults actually had faults [26].

2.1.8.4. Analyzing Design Documents

Furthermore, Stein et al [26] went through a few studies that addressed different perspectives on processing design specifications. One of these studies was conducted by Lague et al where they compared design documents' descriptions of layered architecture systems with the way the source code was organized into files. Another study done by Li and Horgan [26] involved analyzing a design specification to check its correctness before using a tool to automatically generate code from it. They developed a tool called XSuds to go through the design specification, generate a flow diagram, and analyze coverage features of the flow diagram. Then the tool would run a simulation of the design specification, collect the flow data from that, and compare the two sets of flow data. The goal of this study was to facilitate round-trip engineering [26]. Stein et al [26] presented a study done by Lakshminarayana et al. that its goal was to generate visual representations of the metric values for each class in a system, to aid developers in quickly pinpointing areas for improvement. They used Rational Rose's extensibility interface and Rose scripting language to get class information from UML diagrams. They developed a visual representation for each class based on its value for each metric. In the resulting model, the visual representation makes it immediately clear which classes have complicated

interactions with other classes [26]. This allows developers to analyze a large system much more quickly than they ever could with a standard printout of metric values. According to Stein et al [26], Lakshminarayana et al's study is most relevant to their research because both involve processing design specifications to calculate metrics on classes before they get implemented. However, whereas Lakshminarayana et al. processed UML diagrams to compute syntactic metrics, Stein et al. address performing natural language understanding on text in design documents to compute semantic metrics [26].

2.1.9. Metric Extraction Method to Be Visually Represented

One concern in software engineering is how high-quality software can be produced with predictable costs and time. Software metrics include a broad array of measurements for computer software. Metrics can be used in the software development process to help continually improve the software product as it is developed. Software metrics provide a quantitative means to predict the software development process and evaluate the quality of the software products. Several software metrics have been proposed for measurement of structural complexity of procedural software. Examples of these metrics include McCabe's cyclomatic complexity metric and Halstead's software science metric. Lakshminarayana et al [15] presented a new approach to aid understanding of object-oriented software through 3D visualization of software metrics that can be extracted from the design phase of software development. The focus of their paper is a metric extraction method and a new collection of glyphs for multi-dimensional metric visualization. According to Lakshminarayana et al [15], information visualization is a useful tool to aid users in comprehending large and/or complex data. Effective information visualization can accelerate perception and insight into large volumes of data. Scientific visualization, which can be viewed as a branch of information visualization, involves generating complex graphical images representing vast amounts of scientific data derived from real-world physical phenomena in order to help scientists have a better understanding of the data [15]. Another branch of information visualization is software visualization. Software visualization involves the graphical display of software characteristics and behavior. Software visualization techniques can foster better understanding of software

performance or structure. Lakashminarayana et al presented a classification proposed by Price et al [15] that software visualization has two major subclasses, namely, program visualization and algorithm animation. Program visualization is used to visualize static and dynamic characteristics of the program, while algorithm animation is a method to visualize the flow of an algorithm. Software visualization can help software engineers cope with the complexity of large software systems and understand the relationships between the entities, modules, and subsystems in a software system, thereby significantly improving the software quality and its maintainability [15].

2.1.9.1. Software Visualization and Software Metrics

It is often claimed that the object-oriented programming paradigm allows for a faster development time and higher quality software. However, software metrics are less well studied in the object-oriented paradigm. A small number of metrics have been proposed to measure object-oriented systems. One of the first attempts at defining software metrics for object-oriented systems was made by Chidamber and Kemerer. Lakshminarayana et al [15] listed the set of six object-oriented metrics proposed by Chidamber and Kemerer that are based on measurement theory as follows: Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), Weighted Methods per Class (WMC), and the Lack of Cohesion in Methods (LCOM). Moreover, Lakshminarayana et al [15] mentioned another set of six metrics for object-oriented systems that was presented by Li. These include Number of Ancestor Classes (NAC), Number of Descendant Classes (NDC), Number of Local Methods (NLM), Class Method Complexity (CMC), Coupling Through Abstract Data Type (CTA), and Coupling Through Message Passing (CTM) [15].

An important phase in software development using the object-oriented paradigm is the design of classes. Design metrics can aid in assessing class design. For developers, design metrics tend to be more beneficial than metrics of later phases of development. Lakshminarayana et al [15] focus was visually representing design metrics to enhance their utility. Namely, visual representation can assist the software developers in quickly comprehending the values of the metrics and thereby aid detection of anomalies in the design. As a result, the design can be improved and, ideally, made more robust.

2.1.9.2. Software Design Metrics Extraction and Visualization

Lakshminarayana et al [15] grouped seven metrics to use in their tool that were initially proposed by Chidamber and Kemerer and Li. These seven metrics are defined for each class in the design as follows:

1) Depth of Inheritance Tree (DIT), proposed by Chidamber and Kemerer, is the class's depth (level) in the inheritance tree.
2) Number of Children (NOC) for a class is the count of the class's immediate descendents.
3) Number of Ancestor Classes (NAC), proposed by Li, is the count of the ancestor classes in the class inheritance hierarchy.
4) Number of Descendant Classes (NDC) is the count of all the descendant classes (subclasses) of a class.
5) Number of Local Methods (NLM) of a class is the count of local methods which are accessible outside the class (a count of the number of public methods in a class).
6) Coupling of Abstract Data Type (CTA) is the count of classes that are used as abstract data types in the data attribute declaration of a class.
7) Design Coupling through Message Passing (DCTM) is a metric Lakshminarayana et al [15] had created for design-based estimation of the Coupling Through Message Passing (CTM) metric. The DCTM measures the number of objects passed as parameters to the methods of a class [15].

2.1.9.3. Design Metrics Implications and their Visualization

Lakshminarayana et al [15] presented the following implications regarding the various design metrics:

1) DIT: As the DIT value increases, the classes in the lower level of the inheritance tree will inherit many methods. This may lead to potential difficulties when attempting to predict the behavior of a class. Also, the lower a class is in the inheritance tree, the greater is the design complexity. On the other hand, larger values of the DIT metric could imply a higher reusability, since many methods may be reused (through inheritance).

2) NOC: Higher values of NOC imply greater reusability. But this could also lessen the abstraction represented by the parent class (i.e. some of the children might not be appropriate descendants of the parent class). Also, the amount of testing (needed to test each descendant of the parent class) will increase with a higher value of the NOC metric.
3) NAC: This metric represents the influence of parent classes on the class under consideration. A higher value would imply a greater influence. But this would also mean that more testing is required in order to test the operation of class.
4) NDC: Similar to the NOC metric, this metric captures the influence of a parent class on all its descendant classes. The implications are similar to that of the NOC metric. (The NOC metric is a subset of the NDC metric).
5) NLM: This metric indicates the size of a class's interface for other classes. As NLM grows larger (i.e. as the number of local methods increases), more effort is required to comprehend the class's behavior. (Implementation, testing, and maintenance also require more effort).
6) CTA: Coupling is a measure of interconnection between classes. A greater coupling between classes will tend to break the encapsulation provided by the object-oriented paradigm. A higher value of the CTA metric implies that the design is complex; more effort will be necessary to test and maintain the class.
7) DCTM: This metric has similar implications to the CTA metric.

Lakshminarayana et al [15] then presented a UML class diagram that shows the class structure and class relationships. Each box in the following diagram represents a class. There are three compartments in each box - the class name is specified in the first compartment, a list of attributes (with optional types and initial values) are specified in the second compartment, and a list of operations (with optional argument lists and return types) are specified in the last compartment. It is possible to suppress the attribute and operations list to reduce the level of detail in the diagram [15]. Associations represent structural relationships between different classes (not just procedural dependency relationships). These are represented as solid lines between pairs of classes, with the name of the association placed on or adjacent to the association line. Inheritance is

represented by drawing a solid line from subclass to superclass with a triangular arrowhead pointing toward the superclass [15] (as shown in figure 2.4).

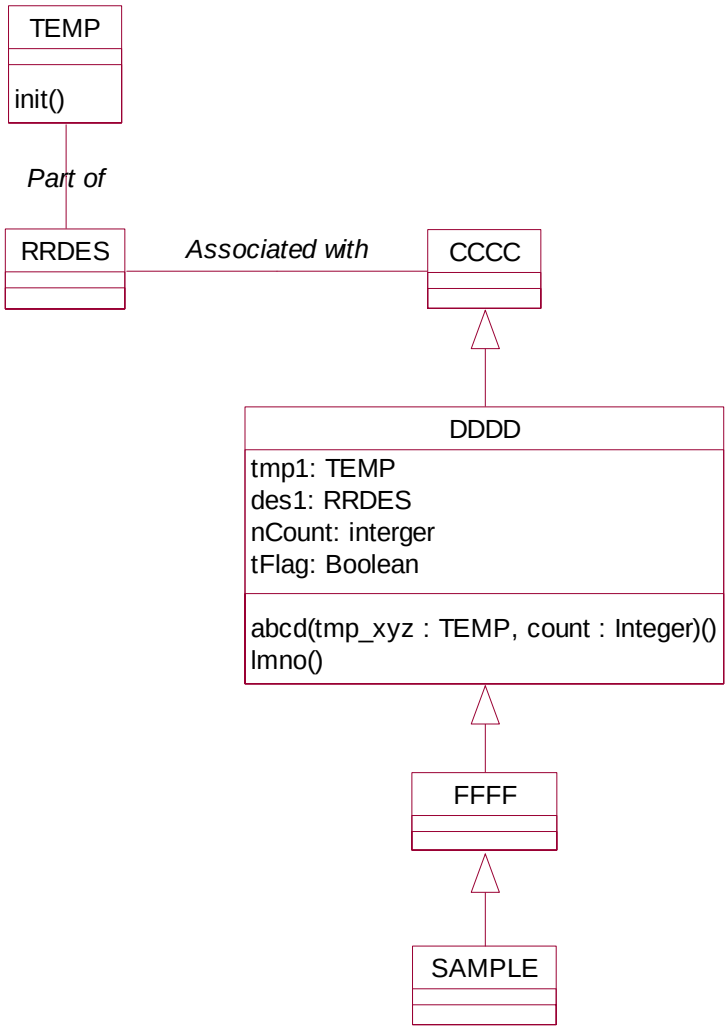


Figure 2.4: UML class diagram (adapted from [15])

A visual representation of the metrics created by Lakshminarayana et al's [15] tool for the classes of the above UML class diagram was displayed. The values of the metrics for a class can be quickly obtained from their visual representation and conclusions about the class complexity can be drawn with ease [15].

2.2. Automation by Software Measurement Tools

This section discusses with some of the automated tools for software measurement. For example, a central repository of an Integrated CASE that records and manages information generated from various CASE tools was developed by Liu et al [18]. Another tool is the one proposed by Tanaka et al [30] that detects and traces irregular size programs or complexly structured programs. Also, Stein et al [26] presented the semMet tool that computes semantic metrics on software systems. Lakshminarayana et al's [15] tool extracts class features and represents the calculated metrics in three-dimensional glyphs.

2.2.1. An Automated Tool for Software Measurement

Liu et al [18] believe that after identifying and defining software quality factors, criteria and the corresponding metrics, the rest of the task is to perform the measurement. In other words, with the Factor-Criteria-Metrics (FCM) model, software quality eventually falls on the software quality measurement. In general, software measurement is a costly task in the absence of an automated tool. Performing software measurement includes the data collection, extraction of measures, and analysis and evaluation [18]. The data collection is concerned with building a software engineering database and recording data of interest from software projects. Data collection is the most labor intensive process in software measurement. The extraction of measures is concerned with purposes of the measurement, and exactly what is to be measured. Determining and defining software metrics are also difficult issues from a theoretical point of view. Due to the expensive labor costs and lack of adequate techniques, software measurement has been a weak part in software engineering [18]. Experiments on the proposed metric measures were carried out for the development process and a research project is concerned with automating software measurement based on an integrated CASE repository was conducted. The prototype developed in this project covers a general OO model so that OO design metrics can be defined and measured. Integrated CASE (ICASE) repositories bring about the opportunity of automating software measurement. ICASE supports requirement analysis, software design and code generation. The central repository of the ICASE records and manages information generated from various CASE tools. The information in the

repository is the resource for software measurement [18]. The repository information is organized by using a meta-model. The meta-model is central to the repository, which describes all models used in each development stage using a uniform scheme. The uniform schema of the meta-model guarantees the share-ability and consistency of the repository information, and facilitates the automation of software measurement. According to Liu et al [18], a metrics tool is required as a part of the integrated CASE to perform the automation of software measurement by navigating repository information. Originally the Oracle CASE supports traditional Functional-Oriented CASE tools such as Entity-Relationship diagram, Data Flow diagram, and Function Hierarchy. The metrics tools in Oracle support COCOMO software cost model, and support the software size model of Function Point Analysis Mark 2 (FPA Mark 2). In Liu et al's research [18], they extended the Oracle CASE repository meta-model to support a general OO design model. Therefore, the tool is able to perform required OO software measurement. In conclusion, they emphasize that the quality of a design is crucial for the quality of software products. To contribute to the control of the quality of OO software products, an FCM model was used to derive possible metrics for OO design. Suggestions were made in considering the metric aspects to produce an OO design. The proposed metric measures were applied to the software development process. A number of complicated experiments to illustrate the approach proposed by Liu et al [18] to developing quality measurement for OO designs were conducted. The general results were satisfactory.

2.2.2. Program Analysis in Parallel with Development Tasks

According to Tanaka et al [30], it is very important to improve software quality by using program analysis and measurement tools and software quality assurance methods at the appropriate points during the process of development. To illustrate, periodic analysis and quality measurements of software products throughout the life - cycle are very important to manage and improve software quality. The importance of program analysis is that repeated analysis in parallel with the actual development phases is the most important point for quality improvement, because analyzed data can be fed back to development in a timely and effective manner. Figure 2.5 shows the process of analyzing programs in parallel with development tasks. Only recently has the development cycle become short;

therefore, some relevant needs became important. For example, to make timely checks of the software quality before unit testing or integration testing. Also, to quickly feedback information about software quality, to improve efficiency of review and testing. Tanaka el al [30] developed a tool that detects and traces irregular size programs or complexly structured programs in which faults tend to appear before testing. The tool also determines review or testing priority and provides messages pointing out concerns about the programs [30].

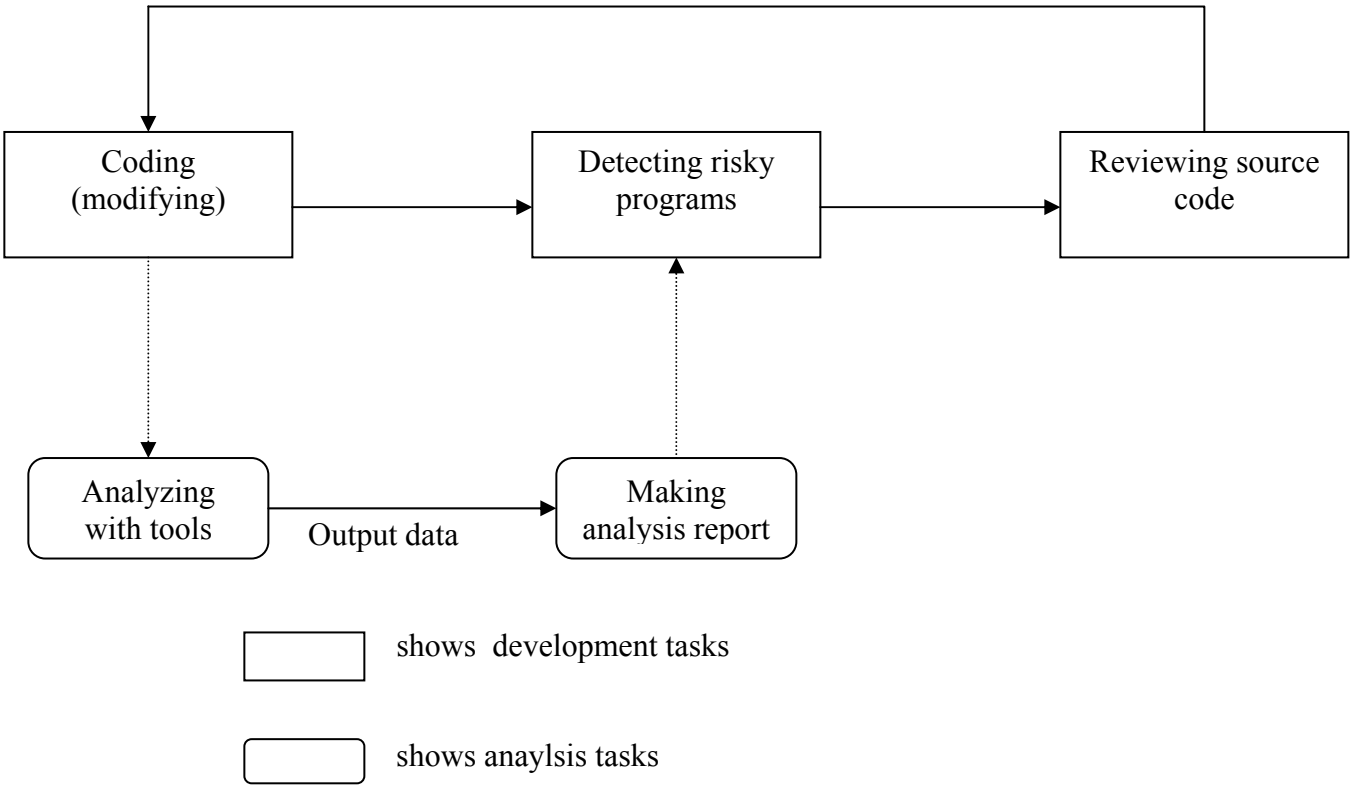


Figure 2.5: Program analysis in parallel with development tasks (adapted from [30])

2.2.3. Computing Metrics on Design Specifications

Stein et al [26] presented a tool that they created called semMet to compute semantic metrics on software systems. In its current form, semMet consists of two parts: the source code interface and the main processing module. The source code interface performs the following steps:

- Generates abstract syntax tree information from code.

- Processes the abstract syntax tree to retrieve the inheritance hierarchy and each class's attributes and behaviors and their accessibility (public, private, or protected).
- Processes the code itself to retrieve all comments at both class and function levels.
- Uses natural language processing to try to determine the part of speech for each identifier. Performs sentence-level natural language processing on comments to determine the part of speech of each word [26].

The main processing module performs the following steps:

- Processes all words from comments and identifiers through a knowledge base of concepts and keywords of the domain of the system.
- Counts concepts and keywords related to each class and each method of each class.

Uses class- and method-level concept and keyword information to calculate metrics and generate a report [26].

These two parts together allow the semMet system to calculate semantic metrics on code. The next step is to modify semMet to calculate semantic metrics from design specifications [26].

2.2.4. Automated Tool for Extraction and Visualization Capabilities

The tool proposed by Lakshminarayana et al [15] includes extraction and visualization capabilities. It can be closely integrated with Computer Aided Software Engineering (CASE) tools. Lakshminarayana et al [15] utilized the Rose script interface in their tool to extract class features which support calculation of the seven design metrics. The visualization tool is aimed at enabling the software developer to obtain a quick understanding of this multi-dimensional information, thereby providing a fast and intuitive means to assess the design complexity and maintainability.

The tool makes use of intuitively meaningful three-dimensional glyphs to represent the ensemble of metrics [15]. For each class in the UML diagram, a 3D object is used to represent the multi-dimensional structural characteristics. Each of these objects begins as

a basic box which is then augmented with glyphs and other cues to allow simultaneous display of all the seven metrics. The following glyphs are used to display the metrics:

1) For the DIT metric, Lakshminarayana et al [15] use a stacking representation along the depth of the glyph box; higher degrees of stacking symbolize a greater depth of inheritance for the class.
2) The number of ancestor classes (NAC) is represented using arrowheads pointing upwards placed on the 3D box. Each full arrow represents a NAC count of two, and half arrow represents a NAC count of one.
3) Arrows pointing downward are used to represent the number of descendant classes (NDC). Each full arrow represents a NDC count of two and a half arrow represents a NDC count of one.
4) The number of children metric (a subset of the NDC metric) is represented using a different coloring (red) for some of the downward arrows. Similar to the NAC and NDC, a half arrow represents a count of one, and each full arrow represents a NOC count of two.
5) The Coupling Through Abstract Data Type (CTA) metric is represented as hooks on the side of the glyph box. The hooks are of three lengths in order to be able to display large values of the CTA metric. Each long hook represents a CTA count of three, and the shortest hook represents a CTA count of one.
6) The Design Coupling Through Message Passing (DCTM) metric is represented by emerging envelopes from the top surface of the glyph (the number of envelopes signifies the DCTM value).
7) The NLM metric is represented by coloring the box boundary. Cold colors (blues) represent a relatively small number of local methods in the class, while hot colors (reds) represent higher counts of local methods [15].

Finally the tool presented by Lakshminarayana et al [15] generates and emits Virtual Reality Modeling Language (VRML) code for the class visualizations and launches an external VRML browser that the developer can use to view the structural characteristics of the design and draw meaningful conclusions.

2.3. Evaluation by Software Measurement Tools

This section explains how each group of researchers evaluated their proposed model. For example, Liu et al [18] chose their metrics based on their experience and understanding in teaching. Others such as Stiglic et al [27] based their work on the findings of a workshop where its main objective was to propose metrics for estimating cost and schedule and for evaluating productivity of OO techniques. Harrison et al [8] based their theoretical validation on a comparison between the results of collecting the MOOD metrics and Kitchenham et al's metrics. Bansiya et al [2] based the validation of the QMOOD quality model on quality attributes' effectiveness and the overall software quality estimation.

2.3.1. Evaluation of Metrics for Object-Oriented Design

Liu et al [18] chose three metrics to indicate the quality of OO design. They selected these metrics on the basis of their experience and understanding in teaching and projects of OO systems development. These three metrics are the number of classes, the degree of interaction between classes, and the length of operations in a class [18]. It was shown that there is a relationship between the total number of classes (TNC), the number of classes newly developed (NCN), and the number of classes reused (NCR) where TNC = NCN + NCR. From this equation, the ratio between NCR and TNC can be expressed as NCR / TNC = 1 – (NCN / TNC) [18]. It was stated that a higher ratio indicates a better quality because of a higher reuse of the classes that have been used and tested in previous cases and that means that these classes are more reliable and correct. Therefore, the number of classes gives us information about the size of a system which varies depending subjectively on the skill and practice of the designer. Liu et al [18] state that a low degree of interaction between classes produces a better quality OO design. The degree of interaction was measured using the average number of message paths per class, which is expressed as TM / TNC where TM is the total number of message paths in the system and TNC is the total number of classes. There are four Chidamber and Kemerer Object Oriented (CK OO) metrics that can be used to assess the degree of interaction between classes [18]. They are depth of inheritance tree, number of children, coupling between object classes, and lack of cohesion in methods. Depth of inheritance of the class is the DIT metric for the class. In cases involving multiple inheritance, the DIT will be the

maximum length from the node to the root of the tree. The deeper a class is in the hierarchy, the greater is the number of methods it is likely to inherit, making it more complex to gain specifications for it. Deeper trees constitute greater design complexity, since more methods and classes are involved which cause difficulty for development. The deeper a particular class is in the hierarchy, the more complicated design steps will be performed on it [18]. Number of children (NOC) calculates the number of immediate sub-classes subordinated to a class in the class hierarchy. The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub-classing. Therefore, it is easy to design those parent classes. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require a more complicated design [18]. As for a class, coupling between object classes (CBO) is a count of the number of other classes to which it is coupled. It relates to the notion that two classes are coupled when methods in one class use methods or instance variables defined by another class. The more independent a class is, the easier it is to extract its object and specifications and to transform it. The larger the number of couplings, the higher the sensitivity to changes in other parts of the design, and therefore development procedures will become more difficult. The higher the inter-object class coupling, the more rigorous design will be added [18]. Lack of cohesion in methods (LCOM) is when considering a class C_1 with n methods $M_{1...}M_n$. Let $\{I_j\}$= set of instance variables used by method M_i. There are n such sets $\{I_1\}\ldots\{I_n\}$. Let $P=\{(I_i, I_j) \mid I_i \cap I_j = \Phi\}$ and $Q=\{(I_i, I_j) \mid I_i \cap I_j \neq \Phi\}$. If all n sets $\{I_1\}\ldots\{I_n\}$ are Φ then let $P=\Phi$. LCOM=$|P| - |Q|$, if $|P| > |Q|$, and = 0 otherwise. Cohesiveness of methods within a class is desirable, as fewer specifications and transformations will be added to the whole program. Lack of cohesion implies classes should probably be split into two or more-sub-classes. Any measure of disparateness of methods helps identify flaws in the design of classes. Low cohesion increases difficulty of design and implementation. The last metric which Liu et al [18] consider particularly relevant to an OO design is the length of operations in a class. Length of operations is usually measured in lines of code (LOCs). At the design stage, this is not applicable. However, a method may call other methods (or itself, if it is recursive) and thus the length of method (LOM) can be defined as the

number of methods called by the method. The length of a method without calling any other method can be counted as one. Each call of the other method adds one to the length. The average length of method per class can be obtained by LOM / TNC. A quality OO design should keep down the average length of operations. Liu et al [18] indicate that one of CK OO metrics can be used here in relation to measuring the length of operations which is the weighted methods per class (WMC). WMC = $\sum C_i$ for i = 1 to n where a class C_1, with methods M_1, … M_n that are defined in the class has c_1,… , c_n as the complexity of the methods. If all method complexities are considered to be unity, then WMC = n, the number of methods. Here the number of methods is calculated as the summation of McCabe's cyclomatic complexity of all local methods [18]. The number of methods and the complexity of methods involved is a predictor of how complex the design will be applied to the software. The larger the number of methods in a class, the greater the potential impact on children since it will be worse in design cases.

2.3.2. Evaluation Criteria for Object-Oriented Software Development

According to Stiglic et al [27], adoption of object-oriented technology by the software industry is to a large extent interfered with a lack of appropriate evaluation criteria. Therefore, they presented a paper to discuss some evaluation criteria, measures and metrics, suitable for object-oriented software development. They believe that the focus of scientific research regarding object orientation has already shifted from implementation to earlier phases of software and information system development [27]. Additionally, the emphasis should be placed on all aspects of software development that have been investigated in the context of structured techniques, from executable specifications, testing strategies to estimation models and metrics. Their work was based on the findings of a workshop whose main objective was to propose metrics for estimating cost and schedule and for evaluating productivity of OO techniques. The main finding, stated at the workshop, was that they have a better insight in product metrics than in process metrics [27].

2.3.2.1. Importance of Measures for OO Software Development

Stiglic et al [27] believe that object technology does not guarantee that the software developed with OO techniques will be better and that the developers have been using the available facilities in the best possible way. Therefore, it is necessary to establish some basic standards and guidelines that developers should follow. Corresponding OO evaluation criteria have to be defined too. These measures should enable objective comparison of the results of accomplished work and provide a reliable evaluation framework. Moreover, if many claimed and expected benefits and advantages of object technology are to be realized and achieved, then measures of OO systems are necessary and inevitable [27]. Automated support might assist investigation and comparison of the achieved and expected benefits as are the improved reusability and higher productivity. Metrics are also important according to the emphasized needs and demands for improvements in the software development process. Measures are necessary to identify weaknesses of the development process. They also direct corrective activities and enable monitoring of the obtained results. In this manner a close loop feedback mechanism is established within which incremental improvements to the software development process can be made over time [27].

2.3.2.2. Proposed Definitions for Measures of OO Software

The authors of "How to Evaluate Object-Oriented Software Development?" paper define the word "metric" as a function, whose value is derived from a product, process, or resource [27]. They stated that it is important to distinguish between objective and subjective metrics. An objective metric is a function whose inputs are software data (elements) and whose output is a single numerical value [27]. Subjective metrics, on the other hand, attempt to track less quantifiable data and usually depend on the subjective judgment. The obtained metric value indicates the degree to which software possesses a given quality attribute. Therefore, quality metrics are an indirect measure of software quality. It is required to have a validated set of metrics, metrics whose values have been proven to be statistically associated with corresponding software attributes. After all, the philosophy of the standard for a Software Quality Metrics Methodology is that an

organization can use whichever metrics it deems most appropriate for its applications as long as the methodology is followed and the metrics are validated [27].

Stiglic et al [27] further discuss the importance of coupling and cohesion for structured approach where they have already proven to be useful criteria for evaluation of the quality of encapsulation. Coupling measures the interface between units. It measures an observed unit's dependence on other units. Cohesion is a qualitative measure that considers the relationship between elements within a unit, how strongly they are connected and how many tasks are performed inside the unit. Ideally, coupling should be minimized and cohesion should be maximized [27].

2.3.3. Evaluation and Comparison of MOOD Metrics with Other Proposed Metrics

Harrison et al [8] present a comparison between the results of collecting the MOOD metrics and Kitchenham et al's metrics for three releases (R1, R2, and R3) of an electronic retail system (ERS) and for the second release of a suite of image processing programs (EFOOP2). These results are shown in the tables 2.7 and 2.8:

	R1	R2	R3	EFOOP2
LOC	1149	2536	2753	8977
Total Classes	4	13	13	12
Total Methods	20	96	96	134
Total attributes	8	35	35	33

Table 2.7: Kitchenham et al's metrics (adapted from [8])

	R1 (%)	R2 (%)	R3 (%)	EFOOP2 (%)
AHF	100	100	100	100
MHF	0	20.4	20.4	6.3
AIF	12.5	0	0	0
MIF	9.1	0	0	0
CF	0	5.8	5.8	3.0
PF	60	Undefined	Undefined	Undefined

Table 2.8: MOOD metrics (adapted from [8])

The Attribute Hiding Factor (AHF) metric for all of these systems has its maximum value of 100 percent, indicating that all the attributes were declared as private. Method Hiding Factor (MHF), on the other hand, has relatively low values, indicating a lack of information hiding. Inheritance was not utilized at all, with the exception of R1 of ERS, as shown by Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF). The undefined Polymorphism Factor (PF) values also reflect this lack of inheritance in the other systems. All of the systems displayed only small amounts of interclass coupling (shown by the Coupling Factor (CF)), possibly pointing to well-designed systems.

Tables 9 and 10 show the MOOD metrics and code metrics, respectively for nine samples of a large commercial retail system [8].

System Label	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)	6 (%)	7 (%)	8 (%)	9 (%)
AHF	45.9	66.7	66.3	44.0	62.5	67.5	52.4	48.5	50.8
MHF	10.1	7.7	16.4	9.5	25.4	15.4	15.8	15.7	15.4
AIF	17.1	11.3	15.3	30.6	46.8	26.1	19.7	36.6	32.0
MIF	15.2	14.3	20.7	27.4	45.5	33.6	22.5	36.5	26.5
CF	3.5	3.5	3.8	6.3	3.1	4.5	5.4	4.9	4.6
PF	4.3	5.4	8.9	2.9	6.7	4.5	6.6	6.2	6.4

Table 2.9: The MOOD metrics (adapted from [8])

System Label	1 (%)	2 (%)	3 (%)	4 (%)	5 (%)	6 (%)	7 (%)	8 (%)	9 (%)
LOC	15837	23750	47106	23154	20747	44930	28582	19254	20085
Total Classes	65	57	91	51	154	92	71	69	74
Total Methods	1446	1535	2141	1420	2814	2224	1978	1815	1876
Total Attributes	537	876	1178	538	1113	1132	839	675	700

Table 2.10: Product metrics (adapted from [8])

From Table 2.8, we can see that the values for the AHF vary between 44 to 68 percent. This is interesting as a figure of 50 percent would suggest an even balance between the public and private data attributes. Ideally, the AHF should be close to 100 percent, to adhere to the concept of information hiding [8]. The values for the Method Hiding Factor (MHF) vary between 8 to 25 percent. These low values indicate a low degree of information hiding, possibly suggesting a lack of abstraction at the design stage.

The Attribute Inheritance Factor (AIF) varies between 11 to 47 percent. These are also rather low, suggesting only a moderate use of inheritance.

The Coupling Factor metric ranges from 3 to 6 percent, suggesting little interclass coupling. According to Harrison et al. [8], Abreu suggests that CF should be neither too low, nor too high. Low coupling reduces potentially harmful side-effects such as unnecessary dependencies and limited reuse. However, a very low value of CF (0 percent) indicates that a system has no interclass coupling, which might point to a pathological system in which classes only communicate via inheritance, or in which there is excessive code duplication. On the other hand, a CF of 100 percent may also indicate a problematic communications infrastructure; excessive coupling implies that software will be difficult to maintain, evolve, and reuse [8].

The values for the Polymorphism Factor metric range from 3 to 9 percent; these low values are fairly typical and unsurprising considering the relatively moderate use of inheritance [8].

Harrison et al [8] conclude that their investigation into the validity of the six MOOD metrics, as far as information hiding, inheritance, coupling, and dynamic binding are concerned, can be shown to be valid measures within the context of this theoretical framework. The main problems which they encountered during their theoretical validation stemmed from imprecise definitions of the attributes to be measured [8].

They believe that the MOOD metrics operate at the systems level. Comparing them with those of Chidamber and Kemerer, the two sets are complementary, offering different assessments of a system. According to Harrison et al [8], the Chidamber and Kemerer metrics appear to be useful to designers and developers of systems, giving them an evaluation of a system at the class level. The MOOD metrics, on the other hand, could be of use to project managers providing an overall assessment of a system [8].

2.3.4. Validation of QMOOD Metrics

Concluding their work, Bansiya et al [2] based the validation of the QMOOD quality model on two levels: validation of the individual quality attributes' effectiveness and validation of the overall software quality estimation. In order to verify that the computed values of the quality attributes are within valid ranges, it was desirable that the quality attribute values be computed for several designs and it was decided that these designs had been developed for similar requirements and objectives. Therefore, Bansiya et al [2] decided to use several versions of two popular Windows application frameworks, Microsoft Foundation Classes (MFC), and Borland Object Windows Library (OWL) as their test-beds. It was expected that the quality characteristics for each version of the two framework systems evaluated should match the generally expected trends from one version to the next. Specifically, it was expected that the quality attributes reusability, flexibility, functionality, extendibility, and effectiveness should increase from one release to the next. Furthermore, Bansiya et al [2] stated that releases of a mature framework are expected to reverse the trend of the understandability measure since development efforts in mature frameworks can be expected to improve their usability, reduce complexity, and make them easier to understand. Using the QMOOD++ tool, Bansiya et al [2] gathered the metric data required and then normalized their values. Then they analyzed and compared these values with the expected results and found that the expected increase in values of the quality attributes is compatible with the hypothesis that these quality attributes should improve with new releases in framework-based systems.

The second level at which Bansiya et al [2] based the validation of the QMOOD quality model was to assess how well the model is able to predict the "overall quality" of an object-oriented software design. They carried out this validation by comparing the predictability of QMOOD for several separate object-oriented designs that had been developed for the same set of requirements with the study of a group of 13 independent evaluators for the same set of object-oriented designs. Bansiya et al [2] ended up finding that the rankings of the validation suite projects indicated a close agreement between the assessments done by the evaluators.

2.3.5. Types of Execution Analysis

As mentioned earlier, Wang [34] classified static analysis, execution analysis, and manual inspection as the three basic metric collection techniques that could be used to assess any software product. Execution analysis is being sub-divided into three types. The first technique in the execution analysis is black-box testing. This is where the executable components are tested against the functional specifications or user manuals using a checklist. It checks for consistency between the documents provided and the actual execution of the code. Black-box testing assesses functionality where the checklist ensures that every function described in the specification is tested and that everything performed by the software is described in the specification. Furthermore, it assesses the usability quality characteristics whereby the standard checks that the functions of the software are described in the user manual. It also checks aspects of the user interface such as if information is presented to the user in a uniform manner and that the error messages produced are useful. Assessing efficiency using black-box testing can be achieved by looking at the response times and comparing them with the performance requirements [34]. The proposed metrics here could be the number of failures discovered during the testing of the product where there are two types of errors: class I are the most significant errors like a system crash or a function not implemented or data corrupted during the test run. If one of these is found the product is deemed to have failed the test. Class II are minor failures such as bad representation of data or error message [34].

Failure data collection is another type for execution analysis where data concerning the type and frequency of failures during execution is being collected. This can be used to ensure that the corresponding faults have been fixed [34]. However, a record of failures cannot be used by itself to assess the software unless we know also the amount of usage the software has had. Failure data collection for software assessment means recording the first manifestation of each failure and the time between the occurrences of each failure. Time in this sense means a measure of the amount of usage rather than of calendar time. Therefore, it assesses reliability where the only way we can estimate the reliability of software (i.e., the probability it will run for a given period of time without failing) is by analyzing the past failure history. The metrics needed with the failure data collection should estimate the times between failures over the amount of usage of the software.

There are various measures of usage time we can use, and the one chosen will depend on the type of software: CPU time, elapsed time (the time between the start and stop of each program run), number of test runs (this may be used if each test run is of similar length in CPU time) [34].

The third type of execution analysis is the test coverage which is a way of measuring the amount of code which has been exercised during testing (either in terms of LOC, number of entities, or number of program branches) [34]. Test coverage contributes to the functionality quality characteristic where it measures the quality of the test data rather than of the code itself. If part of the code has not been tested, then the functionality that this part provides could not have been tested. All coverage metrics are of the form: (number of items executed / total number of items) * 100 and differ only in which items are counted. The most commonly calculated coverage measures are statement coverage, branch coverage (every branch in the control flow-graph), basic block coverage (a basic block is either a single statement or a set of simple statements enclosed in block separators), procedure coverage, and PPP coverage (procedure-to-procedure paths are the edges in the call graph between any two modules) [34].

2.3.6. Checklists as an Example of Manual Inspection

The last basic type of activity that can be used to assess software products is manual inspection where the term inspection covers a whole range of activities based on evaluation of the software by humans [34]. One commonly used approach is by using checklists where they provide a structured way of performing inspection. We can apply inspection to many software components such as function specifications, design documents, user documentation, and source code. Checklists comprise a number of questions, for which there are a finite number of specified replies. Each reply has a score associated with it and adding up the scores will yield a total for that checklist. The checklist score can be viewed as a metric [34]. Inspection via checklists is related to all quality characteristics where it is related to functionality since by applying checklists to the specification and test documentation we can find out if the functions provided by the system are clearly described and if the test data addresses these functions. It is related to reliability where checklists can be used to assess technical aspects of fault tolerance and

recoverability in areas such as restart, rollback, robustness to hardware failure and so forth. It assesses usability where the checklists can check if certain features appear on user documentation such as general description of the product, a table of contents, an index and a list of error messages. Maintainability is being evaluated as checklists can be applied to the source code and design documentation to ensure that the software is easy to understand, such as the meaningful identifier names, and a description of each module. Checklists can check whether those parts which are non-portable have been clearly identified and documented and thus contribute to the portability quality characteristic. Finally, efficiency is being assessed when checklists can be used to identify those software features that will affect efficiency, such as choice of algorithm and optimization of certain parts of the code such as using assembly for the most critical parts [34].

Wang [34] concludes that there is no reason why each of these assessment techniques should be used in isolation. It is often possible to combine two complementary techniques in order to assess a particular characteristic. The tools-based methods such as static analysis and test coverage are quite cheap but many important attributes of the software such as Class I error tracked by black-box testing cannot be measured automatically. Checklists, on the other hand, get around this problem but are labour-intensive. A mixture of the tools-based and inspection-based techniques can lead to a more efficient use of effort during assessment [34].

2.4 Prioritization

We searched extensively for the application of prioritization in the field of quality assessment. Two research papers came close to our work. Their metric sets were not at all similar to ours but they applied prioritization based on weighted values which is what we wanted to apply in our research.

2.4.1 Value - Based

Lee et al [17] provided that Value-based review techniques add cost effectiveness into the review processes and they report on an experiment on Value-based reporting. They consider cost effectiveness as one of the important issues for developing products in a life

cycle. Review is a key activity that can detect defects from the early stage and help in fixing them [17]. The review effectiveness metrics proposed by Lee et al [17] (table 2.11) involved weighted sums of distinct issues reported, using impact metrics. Each issue reported has a priority value and criticality value. Priority values and criticality values have three levels: high, medium, and low. They calculate the effectiveness metric according to the following equation:

$$\text{Effectiveness Metric} = \sum_{\text{issues}} (\text{Artifact Priority}) * (\text{Issue Criticality})$$

Artifact Priority / Issue Criticality	H	M	L
H	9	6	3
M	6	4	2
L	3	2	1

Table 2.11: Review effectiveness metric, Issue metrics, and optimality guidelines (adapted from [17])

The impact of each issue is the product of its priority and criticality value. For example, if one issue has medium priority and high criticality, the impact of the issue is six, the result from two (medium) times three (high). The overall review effectiveness metric is the sum of all the issue impacts [17].

2.4.2 Pair-Wise Comparison

Another reported research on software requirements prioritizing was conducted by Joachim Karlsson [11]. His research was a case study at Ericsson Radio Systems AB of two techniques for software requirements prioritizing as a means for determining the importance of candidate requirements, a pair-wise comparison technique and a numeral assignment technique. In the pair-wise comparison technique, the candidate requirements are compared pair-wise to estimate their relative importance. The scale used by Karlsson [11] for the pair-wise comparisons is outlined in Table 2.12.

Intensity of Importance	Definition
1	Equal importance
3	Moderate importance of one over another
5	Essential or strong importance
7	Very strong importance
9	Extreme importance
2, 4, 6, 8	Intermediate values between the two adjacent judgments
Reciprocals	If requirement *i* has one of the above numbers assigned to it when compared with requirement *j*, then *j* has the reciprocal value when compared with *i*.

Table 2.12: The fundamental scale used for pair-wise comparisons (adapted from [11])

To illustrate the concept of pair-wise comparisons, Karlsson [11] presented an assumed example where there are three candidate requirements; *A*, *B*, and *C* with the following relationships:

- *A* is essentially more important than *B* (intensity of importance 5).
- *C* is moderately more important than *A* (intensity of importance 1/3).
- *C* is very strongly more important than *B* (intensity of importance 1/7).

Accordingly, the relative priorities are to be calculated by inserting the *n* candidate requirements in the rows and columns of a matrix of order *n*. For each pair of requirements, e.g. *A* and *B*, their relative intensity of importance is inserted in the position where the row of *A* meets the column of *B*. In the transposed positions, the reciprocal values of the pair-wise comparisons are inserted. Since a requirement is equally important when compared to itself, a '1' is inserted in the main diagonal. Table 2.13 shows the comparison matrix for the previous example as presented by Karlsson [11]:

	A	B	C
A	1	5	1/3
B	1/5	1	1/7
C	3	7	1

Table 2.13: The comparison matrix for the example proposed by Karlsson (adapted from [11])

Chapter 3: The Solution Approach

In this chapter we describe our solution approach and give a brief explanation about the tool (SDAnalysis) in which we embedded the solution approach. We first summarize the design metrics and acronyms and our assumptions about the relationship between design properties, metrics and quality attributes based on Bansiya et al's [2] set of design metrics in section 3.1. Then we describe our solution approach (the input, the processing and the output) in section 3.2. Finally, we present the tool's architecture and describe how it works in section 3.3.

3.1 The Design Metrics

Bansiya et al [2] used a fairly comprehensive list of design metrics to measure the design properties in a class diagram. The following is a list of the design metrics that we adopted from Bansiya et al's [2] work along with their descriptions as applied in our solution.

Design Size in Classes (DSC): This metric is a count of the total number of classes in the design.

Number of Hierarchies (NOH): This metric is a count of the number of class hierarchies in the design.

Average Number of Ancestors (ANA): This metric value signifies the average number of classes from which a class inherits information. It is computed by determining the number of classes along all paths from the "root" class(es) to all classes in an inheritance structure.

Data Access Metric (DAM): This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class.

Direct Class Coupling (DCC): This metric is a count of the different number of classes that a class is directly related to. This metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.

Cohesion Among Methods of Class (CAM): This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class.

Measure of Aggregation (MOA): This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations whose types are user defined classes.

Measure of Functional Abstraction (MFA): This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class.

Number of Polymorphic Methods (NOP): This metric is a count of the methods that can exhibit polymorphic behavior.

Class Interface Size (CIS): This metric is a count of the number of public methods in a class.

Number of Methods (NOM): This metric is a count of all the methods defined in a class.

Each design metric represents a design property. Table 3.1 [2] shows the relation between the design metrics and the design properties.

Design Property	Derived Design Metric
Design Size	Design Size in Classes (DSC)
Hierarchies	Number of Hierarchies (NOH)
Abstraction	Average Number of Ancestors (ANA)
Encapsulation	Data Access Metric (DAM)
Coupling	Direct Class Coupling (DCC)
Cohesion	Cohesion Among Methods in Class (CAM)
Composition	Measure of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number of Polymorphic Methods (NOP)
Messaging	Class Interface Size (CIS)
Complexity	Number of Methods (NOM)

Table 3.1: Design Metrics Corresponding to Design Properties (adapted from [2])

According to Bansiya et al [2], a quality attribute is a combination of more than one design property. Table 3.2 [2] shows the design properties needed for each quality attribute and their relationships as expressed in index computation equations proposed by Bansiya et al.[2].

Quality Attribute	Index Computation Equation
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation – 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation – 0.33 * Coupling + 0.33 * Cohesion – 0.33 * Polymorphism – 0.33* Complexity – 0.33 * Design size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 * Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction – 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

Table 3.2: Computation Formulas for Quality Attributes (adapted from [2])

3.2 The Solution Approach

3.2.1 The Input: Class Diagram and Raw Metrics

At the outset of the research we hoped to extract the desired list of metrics directly from a class design drawn within a CASE tool. Different versions of Rational Rose (RR) were examined. We expected the extraction of the metrics to be straightforward on RR which would then leave us the tasks of analysis and reporting on the design. However, Rational Rose did not offer the feature of metrics extraction from any drawn design. Hence, we had to work around this problem in 2 steps. The ***first step*** is to draw the design in a CASE tool such as Rational Rose, then import it into another tool to extract the desired list of metrics.

We finally settled on the IBM Rational Software Development Platform version 6.0 as the software for drawing the class diagram and for each diagram, we create a new UML Project. The class diagram is stored in the form of a package holding all the classes inside it with all attributes and operations listed in each class. Also, the relationships between classes are expressed clearly. Finally, we get a complete class diagram file that contains our class diagram information.

This file is then input to the next step in our system. This ***second step*** is a software tool to extract the metrics list of interest from a class diagram. We conducted a wide search to find a measurement tool that imports a class diagram from Rational Rose and collects the same metrics in question. We were directed to the SDMetrics tool (Software Design Metrics tool for the UML) version 2.11 [36] and fortunately, the owner (Jürgen Wüst) of this tool [36] gave us a full free version to use in our academic research. The tool offers its own list of metrics that covers information collected from class, package, object, and composition structure diagrams. The language used for writing the list of metrics offered by SDMetrics [36] is XML. The output of this tool is displayed in the form of a graph or a table view. The output could be exported to different file formats including raw text, which is what we used. We could now export the table view to XML file format for use in the main stage of our system.

We enhanced the XML code for the list of metrics in the SDMetrics package and removed all the metrics that were not within our scope. We developed additional code for the missing metrics and adjusted the code for other metrics that we needed to extract. The list below shows the names and meanings of the metrics used directly from the SDMetrics package:

- NumCls: The number of classes in the package.
- NumAnc: The number of ancestors of the class.
- NumDesc: The number of descendents of the class.
- NumAttr: The number of attributes in the class.
- IC_Attr: The number of attributes in the class having another class or interface as their type.
- H: Relational cohesion.
- NumOps: The number of operations in a class.

There were 3 design metrics required by Bansiya et al [2] for which there are no equivalents in SDMetrics. Therefore, we had to develop the XML code for these 3 design metrics. They are:

- NumPriAttr: The number of private attributes in a class.
- NumPolyMeth: The number of polymorphic methods in a class.

- NumAttrandPara: The number of unique classes that are either attributes' type or parameters' type of methods.

Sample of the Additional Code

The XML code shown below was developed for the 3 missing metrics:

```
1- <metric name="NumPriAttr" domain="class" category="size">
   <description>
        The number of Private Attribute in a class.
   </description>
   <projection relset = "ownedattributes" target = "property" condition = "association="
      and visibility='private'"/>
   </metric>
 2- <metric name="NumPolyMeth" domain="class" category="size">
     <description>
          The number of Polymorphic Methods in a class.
     </description>
     <projection relset = "ownedoperations" condition="name startswith 'virtual'"/>
      </metric>
3- <set name="NumAttrandParaTypeSet" domain="class" mulitset="true">
   <projection relset="AttrTypeSet+ParaTypeSet" />
   </set>
   <metric name="NumAttrandPara" domain="class" category="Coupling (import)">
   <description>The number of unique classes that are either attributes' type or
parameters'
        type of methods.
   </description>
   <compoundmetric term="size(NumAttrandParaTypeSet)" />
   </metric>
```

We modified the code of the metric OpsInh to calculate the sum of inherited operations from distinct classes instead of calculating the total number of inherited operations.

The XML code below shows the modification done to this metric:

```
<metric name="OpsInh" domain="class" category="Inheritance">
<description>The number of inherited operations.((p))
This is calculated as the sum of metric metric://class/NumOps/ taken over
all ancestor classes of the class.
((ul))((li))Also known as NMI ref://LK94/.
((li))See also: metric://class/DIT/.((/ul))
</description>
<projection relset="AncSet" eltype ="class" sum="NumOps" recurse="false"/>
</metric>
```

Finally, we modified the metric:

NumPubOps: The number of public operations in a class

to

NumPriOps: The number of private operations in a class.

The XML code below shows the modification done to this metric:

```
<metric name="NumPriOps" domain="class" category="Size">
<description>The number of private operations in a class.((p))
        Same as metric metric://class/NumOps/, but only counts operations with
        private visibility. Measures the size of the class in terms
        of its private interface.
       ((ul))((li))Also known as: NPM (Number of Private Methods)
</description>
<projection relset="ownedoperations" condition="visibility='private'"/>
</metric>
```

As a final result, the list of design metrics were extracted for any class diagram within the SDMetrics package and saved. The extracted values that are represented in a table view in SDMetrics are now saved in an XML file.

3.2.2 The Processing: Metric and Quality Attribute Calculations

In order to calculate the quality attributes given by Bansiya et al [2] as shown in table 3.2, we had to extract the corresponding design properties from the class diagram. For each design property, there is a design metric as listed in table 3.1. However, the SDMetrics tool [36] does not offer this exact set of design metrics directly. Hence, we had to adjust the list of metrics offered by SDMetrics to obtain the list of metrics of interest. These adjustments were either a calculated value of an arithmetic operation for some metric offered by SDMetrics, or a developed code or a change in parameters in the code given by SDMetrics.

Moreover, we had to make an adjustment to the Reusability, Understandability, and Functionality quality attributes' equations with regards to the sign of the cohesion design property where we reversed its sign from + to -. This issue is explained in section 3.2.3.1.

Table 3.3 summarizes our approach in adjusting the SDMetrics values to make them equivalent to the QMOOD metrics.

QMOOD	Adjusted SDMetrics
DSC: # of classes in the design	NumCls
NOH: # of class hierarchies in the design	Count (NumAnc = 0 && NumDesc > 0)
ANA: average # of classes from which a class inherits info	Sum (NumAnc) / NumCls
DAM: #of private attributes (in class) / Total # of attributes	1- NumPriAttr* (Per Class) / NumAttr (Per Class) 2- Sum No. 1 for all classes / NumCls
DCC: # of classes that a class depend upon	Sum NumAttrandPara* / NumCls
CAM: ∑ of independent set	H
MOA: count of the # of data declarations who are user defined classes	Sum (IC_Attr) / NumCls
MFA: level of nesting of classes in an inheritance hierarchy	1- OpsInh** / OpsInh**+NumOps 2- Sum No. 1 for all classes / NumCls
NOP: Count of polymorphic methods (virtual)	Sum (NumPolyMeth*) / NumCls
CIS: # of public methods in a class	1- (NumOps – NumPriOps**) 2- Sum No. 1 for all classes / NumCls
NOM: # of methods in a class	Sum (NumOps) / NumCls

Table 3.3: Adjusting the SDMetrics' metrics to QMOOD's metrics

3.2.2.1 The Suggested Thresholds for the Design Metrics

In order to analyze the values of the design metrics, and subsequently the quality attributes, we used the Chidamber and Kemerer [5] suite (C&K) as the reference for judging the values of the metrics. According to C&K [5], the following judgments are applied on the NOM, ANA, DCC and CAM:

i. The **NOM** (Number of Methods)

1) The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.

* Metrics adjusted by developing additional code.
** Metric adjusted by changing a parameter in its given code by SDMetrics.

2) Classes with a large number of methods are likely to be more application specific, thus limiting the possibility of reuse.

ii. The **ANA** (Average Number of Ancestors)

1) The deeper a class is in a hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.

2) Deeper trees constitute greater design complexity, since more methods and classes are involved.

3) The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Chidamber and Kemerer [5] gave their viewpoints about the Number of Children (NOC) which is the number of immediate subclasses subordinated to a class in the class hierarchy. While our NOH design metric is a count of the number of class hierarchies in the design, we found that Chidamber and Kemerer's suggestions for NOC do apply to our NOH metric as NOH could be considered as a subset of the NOC. Hence, we interpret their viewpoints to be:

1) The greater the number of children, the greater is the reuse, since inheritance is a form of reuse.

2) The greater the number of children, the greater is the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of sub classing.

iii. The **DCC** (Direct Class Coupling)

1) Excessive coupling between classes is detrimental to modular design and prevents reuse. The more independent is a class, the easier it is to reuse in another application.

2) In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design, and therefore maintenance is more difficult.

3) A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

iv. The **CAM** (Cohesion Among Methods of Class)

1) Cohesiveness of methods within a class is desirable, since it promotes encapsulation.

2) Lack of cohesion implies classes should probably be split into two or more subclasses.

3) Any measure of disparateness of methods helps identify flaws in the design of classes.

4) Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

The last metric judged by Chidamber and Kemerer was the Response For a Class (RFC). However, this metric did not match any of the metrics proposed by Bansiya et al and therefore, we could not apply their judgment for this metric.

In order to judge the extracted metrics, we suggested a threshold for each design metric. However, we faced a problem with this task which is choosing the reference for the threshold. Some design metrics could be related to the number of classes in the whole class diagram. Some others as suggested by Bansyia et al [2] ranged between 0 and 1. Table 3.4 shows a list of the design metrics, our suggested maximum and/or minimum thresholds and some examples.

Design Metrics	Suggested Maximum and/or Minimum Threshold	Examples
DSC	None	None
NOH	NOH = 0 NOH = 1 and ANA > ½ (△DSC/DSC)	If we have *n* classes and each one of them is a stand alone class. If we have *n* classes and all of them under each other (having one root only), then the maximum value of ANA is (△*n*/*n*). Exceeding half this value means that still there is high level of inheritance with reference to the total number of classes.
ANA	ANA = 0 ANA > ½ (△DSC/DSC)	If we have *n* classes and each one of them is a stand alone class. If we have *n* classes and all of them under each other (having one root only), then the maximum value of ANA is (△*n*/*n*). Exceeding half this value means that still there is high level of inheritance with reference to the total number of classes.
DAM	NumPriAttr < (NumAttr/2) (per class)	In a class, if we have *x* private attributes and *y* public attributes, and *x* is less than *y*, then the total number of attributes will be (*x* + *y*). If the number of private attributes is less that half the total number of attributes. Therefore, this will be an offline class.
DCC	NumAttrandPara > (NumAttr + NumOps)/2 (per class)	If a class has *x* attributes of other classes' type and a method with *y* parameters of other classes' type, then the NumAttrandPara will be (*x* + *y*). If this class has a total of *z* attributes and operations which is less than half the value of NumAttrandPara, then this class is an offline class.
CAM	None	None
MOA	IC_Attr > NumAttr/2 (per class)	If a class has *x* attributes of other classes' type and a total of *y* attributes where half the value of *y* is less than *x*, then this class is an offline class.

MFA	NumAnc > 0 and $\frac{\text{OpsInh}}{\text{OpsInh} + \text{NumOps}} < 0.5$	If an inherited class has a total of 7 methods and 4 of them are inherited from the parent class, then the value of MFA for this class will be 4/7 = 0.57 which is logically acceptable. However, if this class inherits only 3 methods and still has a total of 7 methods, then the MFA value will be 3/7 = 0.4 which is not a favorable solution.
NOP	NumPolyMeth > OpsInh/2 (per class)	If a class inherits *x* methods and *y* of them are determined dynamically at run-time and *y* is more than half the value of *x*. Then this offline class will create a problem for the whole structure.
CIS	NumPriOps > NumOps/2 (per class)	If a class has a total of *x* methods and *y* of them are private methods and *y* is more than half the value of *x*; then, this class is an offline one.
NOM	None	None

Table 3.4: The Design Metrics, the Suggested Thresholds, and Examples on the Thresholds

3.2.2.2 The adopted Prioritization scheme

Our survey on prioritization techniques on quality assessment led us to the conclusion that each research devised a prioritization technique based on the problem requirement. For example as mentioned in Chapter 1, Lee et al [17] proposed the effectiveness metrics, whereas Karlsson [11] used the pair-wise comparison technique. We created our own prioritization technique based on a weighted average geometric series which best suits our prioritization requirement. Under this scheme, each quality attribute is assigned a weight according to the following geometric series [29]:

$$w_1 = \frac{1}{\frac{1-(1/2)^n}{1-1/2}}.$$

$$= \frac{1/2}{1-(1/2)^n}. \qquad \text{[Equ. 1]}$$

and

$$w_n = (1/2)^{n-1}\, w_1 \qquad \text{[Equ. 2]}$$

where w_1 is the weight of the first priority and n is the number of priorities chosen by the user. We first calculate the first priority according to Equ.1, and then the following prioritized quality attributes are calculated according to Equ. 2.

For example, considering our six software quality attributes, if the user did not assign any priority to any of them, then they will all get equally-weighted values of 1/6 each. However, if the user gave a priority to each quality attribute, then it will be calculated according to the above equation. For example, if we arrange our quality attributes as follows:

1) Functionality
2) Effectiveness
3) Reusability
4) Flexibility
5) Extendibility
6) Understandability

Then the weighted value to be assigned to Functionality will be 0.51. Then Effectiveness will take the weighted value of 0.25. Reusability will take the weighted value of 0.13. Being the fourth item in the priority list, Flexibility will take the value of 0.06. The weight for Extendibility will be 0.03. Finally, the weight for Understandability will be 0.02. Each weight is multiplied by the value of the corresponding quality attributes. All these multiplications are summed up and displayed as the weighted average value.

It is also possible that the user chooses to assign priorities for only 2 quality attributes and leaves the rest un-prioritized. For example, if the first priority is given to Effectiveness, and the second to be Reusability, and the user selects not to prioritize the rest of the quality attributes list then the calculation for the weighted average for each priority will be as follows:

Effectiveness will take the first weight of 0.57. Then Reusability will take the weight of 0.29, and then each of the remaining quality attributes will take an equal weight of 0.14. Finally the weighted average value is calculated as the sum of the products of each computed quality attribute value multiplied by its weight.

3.2.3 The Output: Observation and Analysis Report

The aim of the observation report is to identify the deficiencies in the quality of the class diagram being examined. The report is based on Bansiya et al's quality metrics suite that we selected to work with (as listed earlier in section 3.2.2). Within the report, a threshold for each quality metric is first generated (as explained in section 3.2.2.1 above). Some thresholds are computed for each individual class (e.g. direct class coupling) while some others are computed for the entire class diagram (e.g. number of hierarchies). We then test the selected metric values for the given class design against the corresponding threshold values. We alert the user whenever a metric value generated from the class diagram deviates from its threshold values. We also identify the sources of deviation and may even suggest some solutions to the user.

From within the tool, we present a more detailed observation report to the user. The user would then know the exact classes that are causing the defect in his/her design and may be guided to fix the defect without restructuring the whole class diagram. Hence, our tool does not only help the user assess the quality of the class diagram, but also identifies the possible sources of deficiencies and in many cases can help direct the user as to how he/she can go about treating the deficiencies.

3.2.3.1 Analysis of Specific Metrics

In the next few paragraphs we summarize our approach in addressing the design metrics suggested by Bansiya et al [2]. Our method in handling each metric is explained separately under the metric name. It is worth noting here that all values that are transformed from decimal to whole numbers are based on the floor of the computed values.

DSC

Design Size in Classes (DSC) is a count of the total number of classes in the design. According to Bansiya et al's [2] experimental work, the total number of classes in one package could reach up to 356 classes as in the Object Windows Library, OWL 5.2 project that was released mid 1997 with Borland C++ 5.2. Therefore, it is difficult to set a threshold for this metric.

NOH

The Number of Hierarchies (NOH) metric measures the number of class hierarchies in the design. It counts the number of non-inherited (root) classes that have children in the design. Two extreme cases are considered as worst case scenarios. The first extreme case is when all classes are totally disjoint and each one stands alone (as shown in figure 3.5 if we have 7 classes in the diagram). The other extreme case is when all classes are under one root (as shown in figure 3.6, with 7 classes). If we calculate the value of NOH for the fist scenario, we will get a zero as there are no hierarchies. But, if we calculate the value of NOH for the second scenario, we will get the value of 1 as there is only one hierarchy. All other cases would give a value greater than 1. If we calculate the maximum NOH value for the 7 classes, we will get 3 which is the floor value of the total number of classes divided by 2. Therefore, the range of NOH is between 0 and $\lfloor DSC/2 \rfloor$.

If the value of NOH is 0, we inform the user that the design lacks hierarchy as classes are disjoint and each one stands alone.

Also, if the value of NOH is equal to 1, we check if the value of ANA is the maximum value (see ANA below for full definition of the maximum value), then we inform the user that the design consists of only one hierarchy and that there is high dependency between classes.

The above two extreme cases negatively affect the functionality quality attribute of the whole design.

Figures 3.1 through 3.6 show sample diagrams for acceptable and unacceptable class inheritances and their corresponding ANA and NOH values.

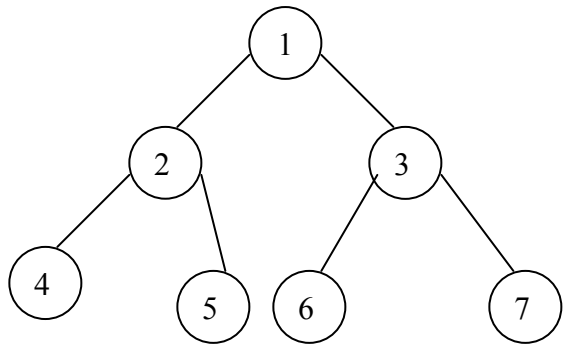


Figure 3.1: First Acceptable Class Diagram that consists of 7 classes, ANA=10/7=1.43, NOH = 1

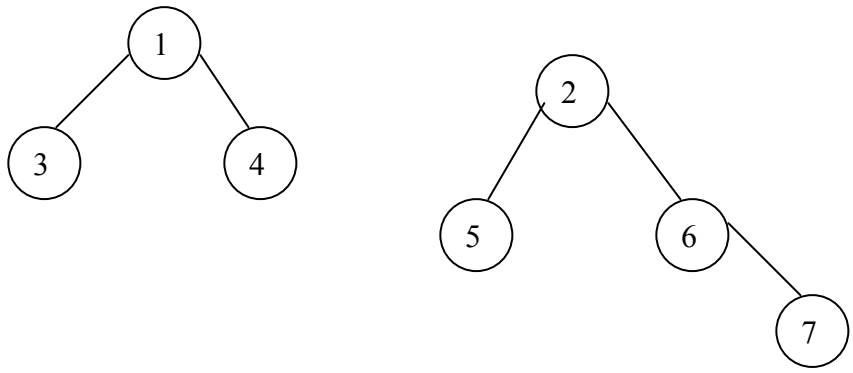


Figure 3.2: Second Acceptable Class Diagram that consists of 7 classes, ANA=6/7=0.86, NOH = 2

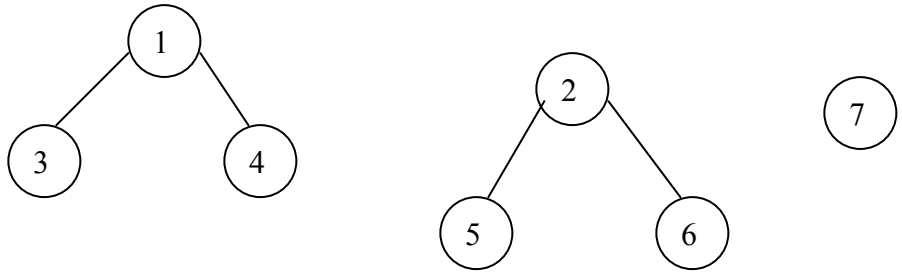


Figure 3.3: Third Acceptable Class Diagram that consists of 7 classes, ANA=4/7=0.57, NOH = 2

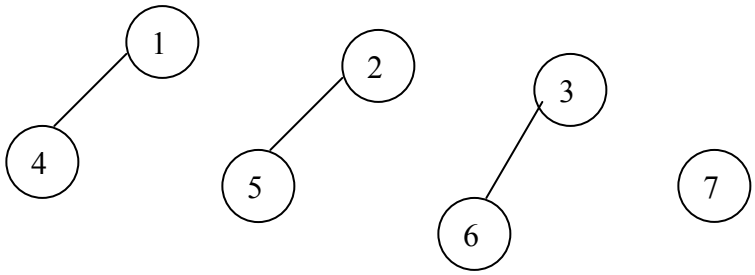


Figure 3.4: Class Diagram that consists of 7 classes resulting in max. value for NOH, ANA= 3/7 = 0.43, NOH = 3

Figure 3.5: First Worst Case Scenario for 7 Classes that are Totally Disjoint, ANA = 0, NOH = 0

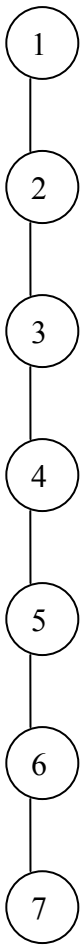


Figure 3.6: Second Worst Case Scenario for 7 Classes that are Under One Root, ANA = 21/7 = 3, NOH = 1

ANA

The Average Number of Ancestors (ANA) is computed by determining the number of classes along all paths from the "root" class(es) to all classes in an inheritance structure. Similar to NOH, we have the same two worst case scenarios where the first case is the disjoint classes shown in figure 3.5 and the other case is when they are all under one root as shown in figure 3.6. When we have all classes under one root (figure 3.6), this will give us the maximum value for ANA which is the triangular number [35] of DSC divided by the total number of classes (DSC). The triangular number for *n* classes is the sum of ancestors from 1 to (*n* - 1) which is computed as (*n**(*n* - 1))/ 2 [35]. However, if we examine the first worst case scenario where there is no class inheritance (figure 3.5), then in this case the value of ANA will be zero. Therefore, the threshold that we suggest for

this metric is either ANA equal to zero or ANA greater than (1/2(△DSC/DSC)). For example, in the case of the class diagram that consists of 7 classes, the range of values is between 0 and 3. Consequently, we compare the value of the ANA to the threshold and if the computed ANA is equal to 0, we inform the user that the design exhibits a bad inheritance structure where there is no inheritance which is undesirable. Also, if the computed ANA is greater than half the maximum value of ANA, we similarly alert the user that the design consists of a huge hierarchy structure which in turn means that there is no abstraction. We indicate that this will negatively affect the extendibility and effectiveness of the overall design.

DAM

The Data Access Metric (DAM) is the ratio of the number of private attributes to the total number of attributes declared in a class. Bansiya et al [2] mentioned that a higher value for DAM is desired as this metric will highly enhance the encapsulation design property. Accordingly, it is preferable to have a high average number of private attributes to the total number of attributes per class. We therefore consider any class that has a number of private attributes which is less than half the total number of attributes as being a weak class and that this in return will affect the overall value for this metric. We report that the specific class is offline regarding the number of private attributes to the total number of attributes and that it negatively affects the overall flexibility, understandability, and effectiveness quality attributes of the whole class diagram.

DCC

Direct Class Coupling (DCC) is a count of the different number of classes that a class is directly related to in the form of attribute declarations or message passing in methods. For each class, we count the number of unique classes that are either attribute type or parameter type within a method. If this counted value exceeds half the value of the total number of attributes and operations for the same class, we alert the user about this class. We display the class name and inform the user that this class exhibits high coupling.

CAM

Bansiya et al [2] define the computation of the metric Cohesion Among Methods (CAM) for a Class to be the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in a class. However, we found a readily

calculated value for the cohesion metric in SDMetrics [36] based on the average number of internal relationships per class. According to SDMetrics owner, Juergen Wuest, cohesion is the degree to which the elements in a design unit (package, class etc) are logically related. He explains the proposed cohesion metric as quantifying the connectivity between elements of the design unit: the higher the connectivity between elements the higher the cohesion. Wuest continues to explain how previously proposed cohesion metrics are normalized to have a notion of minimum and maximum cohesion, usually expressed on a scale from 0 to 1. Minimum cohesion (0) is assumed when the elements are entirely unconnected, maximum cohesion (1) is assumed when each element is connected to every other element. Finally, Wuest introduces the idea of not normalized metrics which are based on counts of connections between design elements in a unit (e.g., method calls within a class). As such, un-normalized metrics are conceptually similar to complexity metrics. Therefore, according to Wuest a low cohesive design element has been assigned many unrelated responsibilities. Consequently, the design element is more difficult to understand and therefore also harder to maintain and reuse. Design elements with low cohesion should be considered for re-factoring, for instance, by extracting parts of the functionality to separate classes with clearly defined responsibilities. Therefore, we used the metric proposed by Wuest, which is called H, as a measure for the CAM metric. The importance of this metric is that it adversely affects the reusability, understandability, and functionality of any class diagram and hence in the corresponding equations, we reversed the sign assigned to cohesion (from + to -).

MOA

Measure of Aggregation (MOA) is a count of the number of data declarations whose types are user defined classes. We compare the total number of attributes which exhibit this feature to the average total number of attributes per class. If the compared value exceeds half the total number of attributes, we inform the user that this class has more than half of their data declaration types as user defined classes and that this in turn negatively affects the composition design property and the flexibility quality attribute of the whole design.

MFA

The Measure of Functional Abstraction (MFA) is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. Hence, the normal range of values for this ratio is between 0 and 1. In each descendant class, we check if the value of MFA for this class is less than 0.5, then we alert the user that the total number of methods for this class by far outnumbers the inherited methods, which is structurally fault-prone and that this will affect the values of extendibility and effectiveness quality attributes. In each class we check if the number of methods is larger than twice the number of inherited methods, then we alert the user that this structure can negatively affect all quality attributes. This measure is significant in determining the inheritance feature.

NOP

The Number of Polymorphic Methods (NOP) counts the methods that can exhibit polymorphic behavior. It is a measure of services that are dynamically determined at run-time in an object. We set our threshold for this metric to be not more than half the total number of inherited methods. Our justification is that the polymorphism design property directly affects the flexibility, functionality, extendibility, and effectiveness quality attributes of the whole design. Therefore, the user needs to know which classes with numerous virtual methods will dynamically match during run-time. For example, if a class inherits 6 methods and 5 of these methods are virtual, then this will cause enormous overhead during run-time.

CIS

The Class Interface Size (CIS) metric measures the number of public methods in a class. It is a measure of services that a class provides to other classes. Therefore, the more the public methods found in a class the better is the overall result of this metric. We check if the number of private methods is greater than half the total number of methods in the class. We inform the user that this class is offline and accordingly reusability and functionality quality attributes will be negatively affected.

NOM

The Number of Methods (NOM) is a count of all the methods defined in a class. It is an important metric as it measures the degree of difficulty in understanding and

comprehending the internal and external structures of classes and their relationships. The NOM metric measures the complexity of any design and hence, the more complex a design, the harder it is to understand. However, it is very hard to suggest a threshold for this metric as there is no logical value for the number of operations per class. Also, this was very clear in the examples cited by Bansiya el al [2] where a design with 92 classes has 9384 methods, i.e. 102 methods per class.

Table 3.5 summarizes our suggested maximum and/or minimum thresholds for each design metric and the messages displayed to the user in the case of deviation from the threshold.

Design Metrics	**Suggested Maximum and/or Minimum Threshold**	**Message to the User (in case of deviation)**
DSC	None	None
NOH	NOH = 0 NOH = 1 and ANA > ½ (△DSC/DSC)	The design lacks class hierarchy as classes are disjoint and each one stands alone. This negatively affects the functionality quality attribute of the whole design. The design consists of one hierarchy structure and there is high dependency between classes. This negatively affects the functionality quality attribute of the whole design.
ANA	ANA = 0 ANA > ½ (△DSC/DSC)	The design lacks class hierarchy as classes are disjoint and each one stands alone. This means that there is no abstraction and in return extendibility and effectiveness quality attributes are negatively affected. The design consists of a huge hierarchy structure which means that there is no abstraction. This in turn affects the extendibility and effectiveness of the overall design.
DAM	NumPriAttr < (NumAttr/2) (per class)	*"Classes' Names"* classes have more public attributes than private attributes and this is structurally unfavorable. Their encapsulation values are low and therefore, they can affect the overall values of flexibility, effectiveness, and understandability quality attributes.

DCC	NumAttrandPara > (NumAttr + NumOps)/2 (per class)	*"Classes' Names"* classes have high coupling and need restructuring to decrease the number of relatedness between objects whether as attribute declarations or message passing in methods.
CAM	None	None
MOA	IC_Attr > NumAttr/2 (per class)	*"Classes' Names"* classes have more than half of their data declarations' types as user defined classes and therefore, they affect the composition factor of the overall design.
MFA	NumAnc > 0 and $\frac{\text{OpsInh}}{\text{OpsInh + NumOps}} < 0.5$	*"Classes' Names"* are descendent classes; however, their own methods by far outnumber what they inherit from their parent classes. This negatively affects the extendibility and effectiveness quality attributes.
NOP	NumPolyMeth > OpsInh/2 (per class)	*"Classes' Names"* classes have too many virtual methods. This excess in the polymorphic behavior of the design makes it harder to understand.
CIS	NumPriOps > NumOps/2 (per class)	*"Classes' Names"* classes have more private methods than public ones. Therefore, their methods are not accessible for other classes leading to higher independency between classes.
NOM	None	None

Table 3.5: Summary of Design Metrics and Corresponding Messages for each Offline Metric

3.3 The SDAnalysis Tool

3.3.1 The SDAnalysis Architecture

The architecture of the SDAanlysis tool is shown in figure 3.9. Its main components are the user's visual interface, XML file reader/extractor, the metrics and quality attributes calculator and the report generator. In Appendix A we give a more detailed description of the SDAnalysis tool classes.

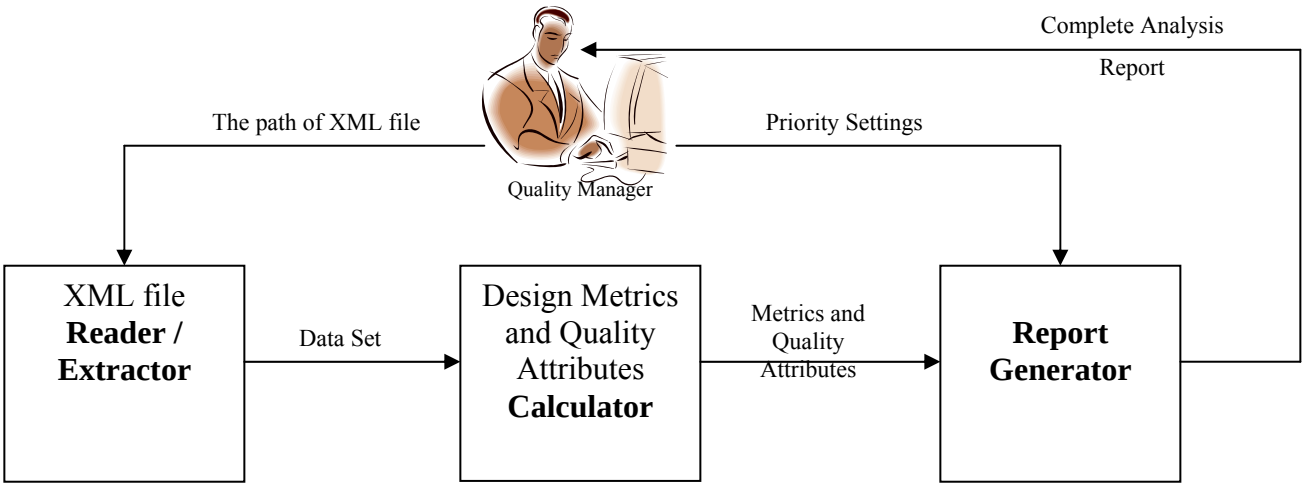


Figure 3.7: The architecture of the SDAnalaysis Tool

3.3.2 How the Tool Works

The SDAnalyis tool implements our solution approach which we described above in section 3.2. The tool works as follows:

1) Initializes all parameters that will hold the values of metrics to zero.

2) Displays the main window for the user to upload the xml file.

3) Checks the path of the xml file and if there is a mistake in the path, displays an error message.

4) Reads the data in the xml file.

5) Calls the CalculateTotal () function to execute the following steps:

a. For every metric collected from the SDMetrics, sums up all its values across all classes.

b. For each design metric presented by Bansiya et al [2], computes an average by dividing the above summed value (in a) by the total number of classes.

6) Defines the required design metrics along with their weights for each quality attribute according to the equations shown in Table 3.2 (Default).

7) Displays the main window once again to the user, to arrange the quality attributes in priority form according to the user's selections (if he/she chooses to prioritize).

8) Reads in the priority as entered by the user and assigns a priority value to each quality attribute according to the following geometric series [29]:

$$w_1 = \frac{1}{\frac{1-(1/2)^n}{1-1/2}} = \frac{1/2}{1-(1/2)^n}. \qquad \text{[Equ. 1]}$$

and

$$w_n = (1/2)^{n-1}\, w_1 \qquad \text{[Equ. 2]}$$

where w_1 is the weight of the first priority and n is the number of priorities chosen by the user. The first priority quality attribute is calculated according to Equ.1, and then the following prioritized quality attributes are calculated according to Equ. 2.

9) Calculates the weighted average for all the 6 quality attributes according to the weight assigned by the user to each of them.

10) Displays the weighted average along with all the values of the 11 metrics outlined by Bansiya et al [2]. A tab shows the individual values of the 6 quality attributes. Another tab displays the analysis report based on our suggested thresholds.

Chapter 4: Experimental Tests and Results

In this chapter, we present 3 examples of test cases on 3 different class diagrams. We show the priority settings and analysis report generated in each case which is fairly distinct from the other two. The reports demonstrate our suggested solution approach (as described in chapter 3 and embedded in the SDAnalysis tool) for computing and addressing weak design metrics in class diagrams and their associated quality attributes. The first two examples span a wide variety of weaknesses in class metrics. The third example focuses on the hierarchy and abstraction design properties as they are very much linked to each other. Each example is fully covered in one section of the chapter.

4.1 The First Example

In this section, we present our first example of an adapted class diagram [31] with some weak design attributes and we show how our analysis report can assist in correcting this class diagram. The initial class diagram is shown in figure 4.1.

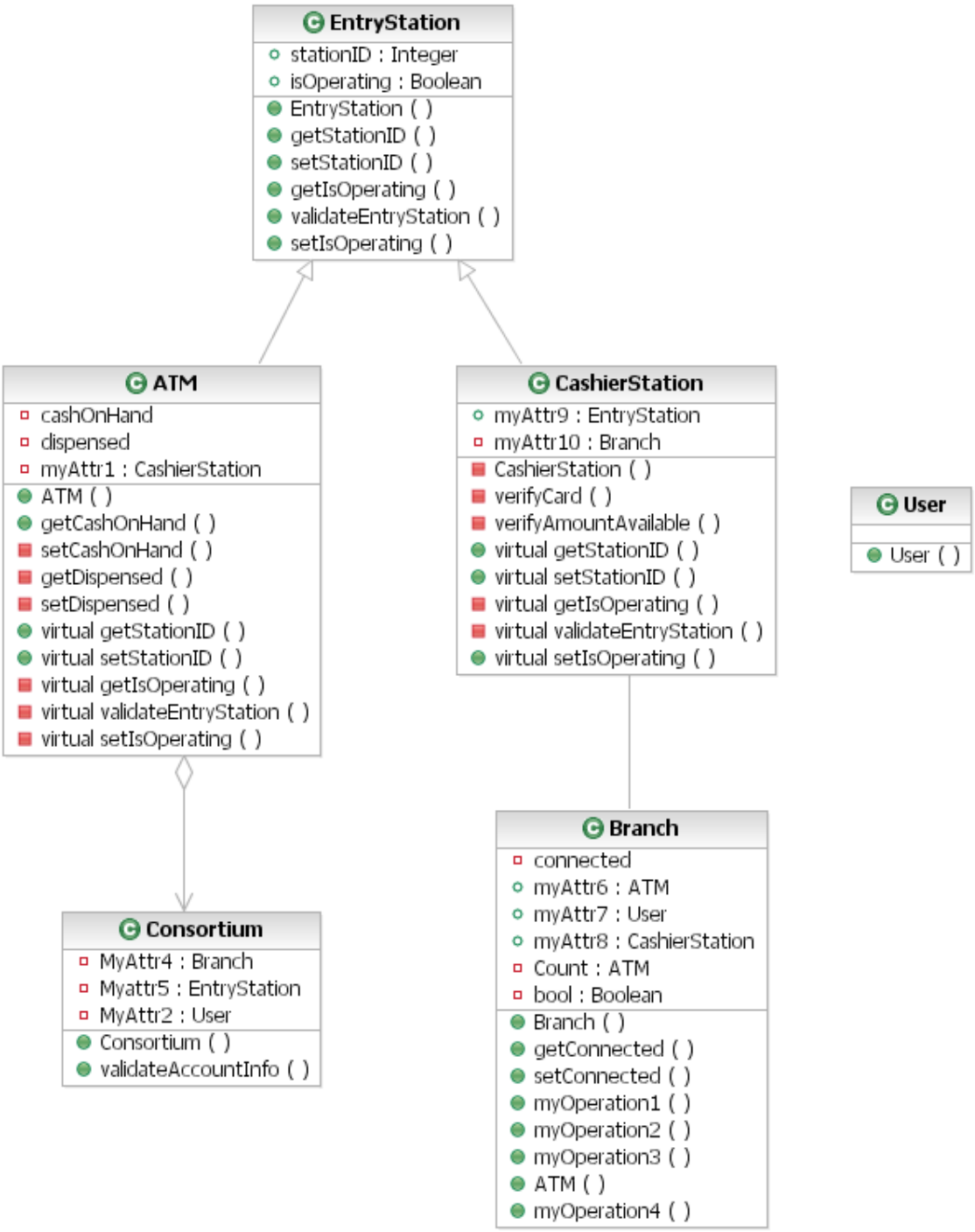


Figure 4.1: First Adapted Class Diagram [31]

We applied 3 cases of priority settings on this example. In the first case **(case I)** we applied an equal priority scheme for the quality attributes: Reusability, Functionality, Flexibility, Extendibility, Effectiveness and Understandability.

In the second case **(case II)** we gave each quality attribute a separate priority as follows:

Priority 1 : Reusability

Priority 2 : Functionality

Priority 3 : Flexibility

Priority 4 : Extendibility

Priority 5 : Effectiveness

Priority 6 : Understandability

In the third case **(case III)** we gave two quality attributes a higher priority and the remaining attributes were given equal priorities. The priority settings were set as follows:

Priority 1 : Extendibility

Priority 2 : Functionality

Priority 3 : Reusability, Flexibility, Understandability, Effectiveness

Table 4.1 shows the set of design metrics and their values as computed for the class diagram in figure 4.1. These values were calculated by both the SDMetrics and SDAnalysis tools.

Design Metric	**Value**
DSC (Design Size)	6
NOH (Hierarchies)	1
ANA (Abstraction)	0.33
DAM (Encapsulation)	0.5
DCC (Coupling)	2
CAM (Cohesion)	3.17
MOA (Composition)	1.5
MFA (Inheritance)	0.13
NOP (Polymorphism)	1.67
CIS (Messaging)	4
NOM (Complexity)	5.83

Table 4.1: The values for the design metrics for the adapted class diagram

The SDAnalysis tool calculated the quality attributes according to the formulas given by Bansiya et al. In Table 4.2a we show the values of these quality attributes for the class diagram in figure 4.1 and Table 4.2b shows the weights obtained for the 3 cases of priority settings.

Quality Attribute	Value
Reusability	3.71
Flexibility	1.21
Understandability	-6.10
Functionality	2.41
Extendibility	0.06
Effectiveness	0.83

Table 4.2a: The quality attributes values for the adapted class diagram

Case #	Weighted Average Value
Case I	0.35
Case II	2.58
Case III	0.66

Table 4.2b: The weighted averages obtained for the 3 cases of priority settings

The SDAnalysis tool computes the design metrics for each class separately. Our aim is to identify the classes that have drawbacks and to decide whether they do affect the whole class diagram metrics (and consequently the quality attributes).

The analysis report generated from the SDAnalysis tool for the class diagram in figure 4.1 included a number of observations which were as follows:

Observation No. 1:

EntryStation classes have more public attributes than private attributes and this is structurally unfavorable. Their encapsulation values are low and therefore, they can affect the overall values of flexibility, effectiveness, and understandability quality attributes.

Observation No. 2:

Consortium classes have high coupling and need restructuring to decrease the number of relatedness between objects whether as attribute declarations or message passing in methods.

Observation No. 3:

CashierStation, and Consortium classes have more than half of their data declarations' types as user defined classes and therefore, they affect the composition factor of the overall design.

Observation No.4:

ATM and CashierStation are descendent classes; however, their own methods by far outnumber what they inherit from their parent classes. This negatively affects the extendibility and effectiveness quality attributes.

Observation No. 5:

CashierStation and ATM classes have too many virtual methods. This excess in the polymorphic behavior of the design makes it harder to understand.

Observation No. 6:

ATM and CashierStation classes have more private methods than public ones. Therefore, their methods are not accessible for other classes leading to higher dependency between classes.

Each class was revised separately and the defects detected and repaired as follows:

1. The class EntryStation contains two attributes that are not used by any other class; therefore, there is no need to have them as public attributes. As a result, we changed their visibility to private.
2. The values for the Direct Class Coupling (DCC) and Measure of Aggregation (MOA) metrics for the Consortium class were observed to be very high. We noted that this class has three attributes whose types are defined to be of other classes. According to Basili et al [3] highly coupled classes are more fault-prone than weakly coupled classes because they depend more heavily on methods and objects defined in other classes. Hence, the Consortium class metrics indicate that it has high coupling. A close examination of the class attributes revealed that two of

3. The CashierStation class was found to have high values for the Measure of Aggregation (MOA), Number of Polymorphic Methods (NOP), Measure of Functional Abstraction (MFA), and Class Interface Size (CIS) metrics. We noted that this class has two attributes declared to be of other class types; has five virtual methods out of a total of eight methods; and has five private methods out of the same eight methods. Moreover, we found that this class returns 3 primitive types, namely; Float, Integer, and Boolean, but there is no attribute declarations for these types. Therefore, we declared 3 new attributes of these types. The private methods were carefully studied and we found that there is no need to have them as private methods and hence we changed their visibility to public. However, with respect to the virtually inherited methods we found that all five methods need to be substituted at run-time as their services are dynamically determined. It was difficult to alter any of them and hence we could not handle the observations about the MFA and NOP metrics.
4. The ATM class is similar to the CashierStation class in that it exhibits high values for MFA, NOP and CIS metrics. The solutions applied to the CashierStation class were also applied to the ATM class where the private methods' visibility was changed from private to public and the virtually inherited methods were left unchanged.

The above solutions were implemented based on the analysis report. After repairing the identified errors that were detected in the observations, the class diagram was restructured. The corrected diagram is shown in figure 4.2.

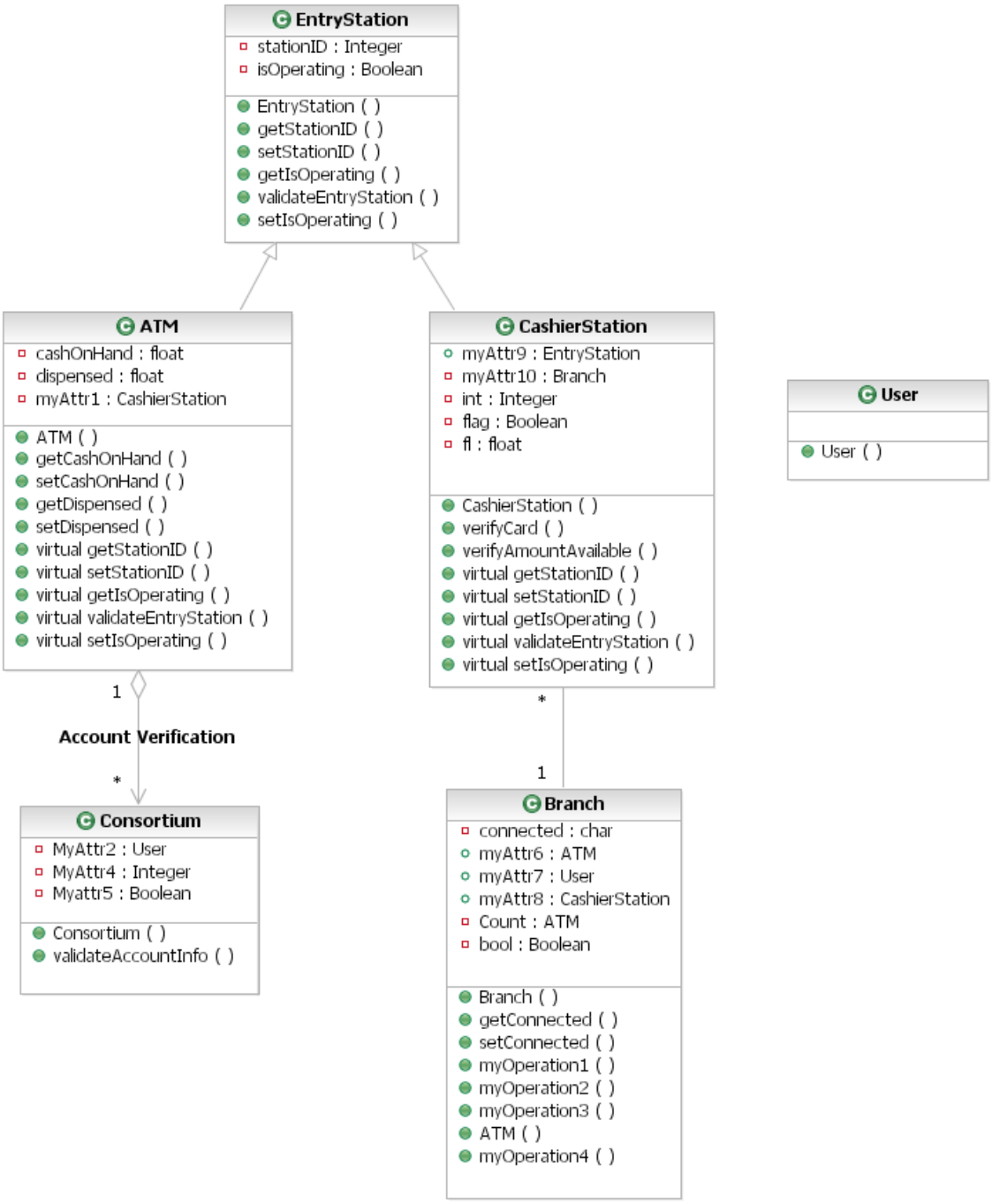


Figure 4.2: The Refined Class Diagram of Figure 4.1

We ran the corresponding file for the diagram in figure 4.2 into SDMetrics to get the required metrics and the results were piped into the SDAnalysis tool. Table 4.3 shows all the values extracted from the class diagram by SDMetrics and calculated by the SDAnalysis tool.

Design Metric	Value
DSC (Design Size)	6
NOH (Hierarchies)	1
ANA (Abstraction)	0.33
DAM (Encapsulation)	0.72
DCC (Coupling)	1.33
CAM (Cohesion)	2.83
MOA (Composition)	1.17
MFA (Inheritance)	0.13
NOP (Polymorphism)	1.67
CIS (Messaging)	5.83
NOM (Complexity)	5.83

Table 4.3: Design Metrics Values for the refined class diagram

Table 4.4a shows the resulting quality attribute values for the refined class diagram. Table 4.4b shows the weights obtained for the 3 cases of priority settings.

Quality Attribute	Value
Reusability	4.88
Flexibility	1.27
Understandability	-5.70
Functionality	2.85
Extendibility	0.40
Effectiveness	0.80

Table 4.4a: Quality attribute values for the refined class diagram

Case #	Weighted Average Value
Case I	0.75
Case II	3.33
Case III	1.20

Table 4.4b: The weighted averages obtained for the 3 cases of priority settings

The observations in the analysis report generated by SDanalysis on the refined class diagram in figure 4.2 were as follows:

Observation No. 1:

ATM and CashierStation are descendent classes; however, their own methods by far outnumber what they inherit from their parent classes. This negatively affects the extendibility and effectiveness quality attributes.

Observation No. 2:

CashierStation and ATM classes have too many virtual methods. This excess in the polymorphic behavior of the design makes it harder to understand.

As mentioned above, it was important to keep the polymorphic operations in both the ATM and CashierStation classes to be dynamically determined at run-time. These are the operations that correspond to the overridden operations in the EntryStation class such as getStationID(), setStationID(), getIsOperating(), setIsOperating(). However, we show in the next paragraph the effect of removing the overridden operations from the parent class and retaining them in the child classes.

For the sake of the experiment, we removed the methods in the EntryStation class that were redefined in both ATM and CashierStation classes and removed the word "virtual" from these methods in the previous two (ATM and CashierStation) classes. Figure 4.3 shows the new class diagram after these changes.

Typically, we ran the corresponding file for the diagram in figure 4.3 into SDMetrics to get the new metrics and the results were piped into the SDAnalysis tool. Table 4.5 shows all the design metric values extracted from the class diagram by SDMetrics and calculated by the SDAnalysis tool.

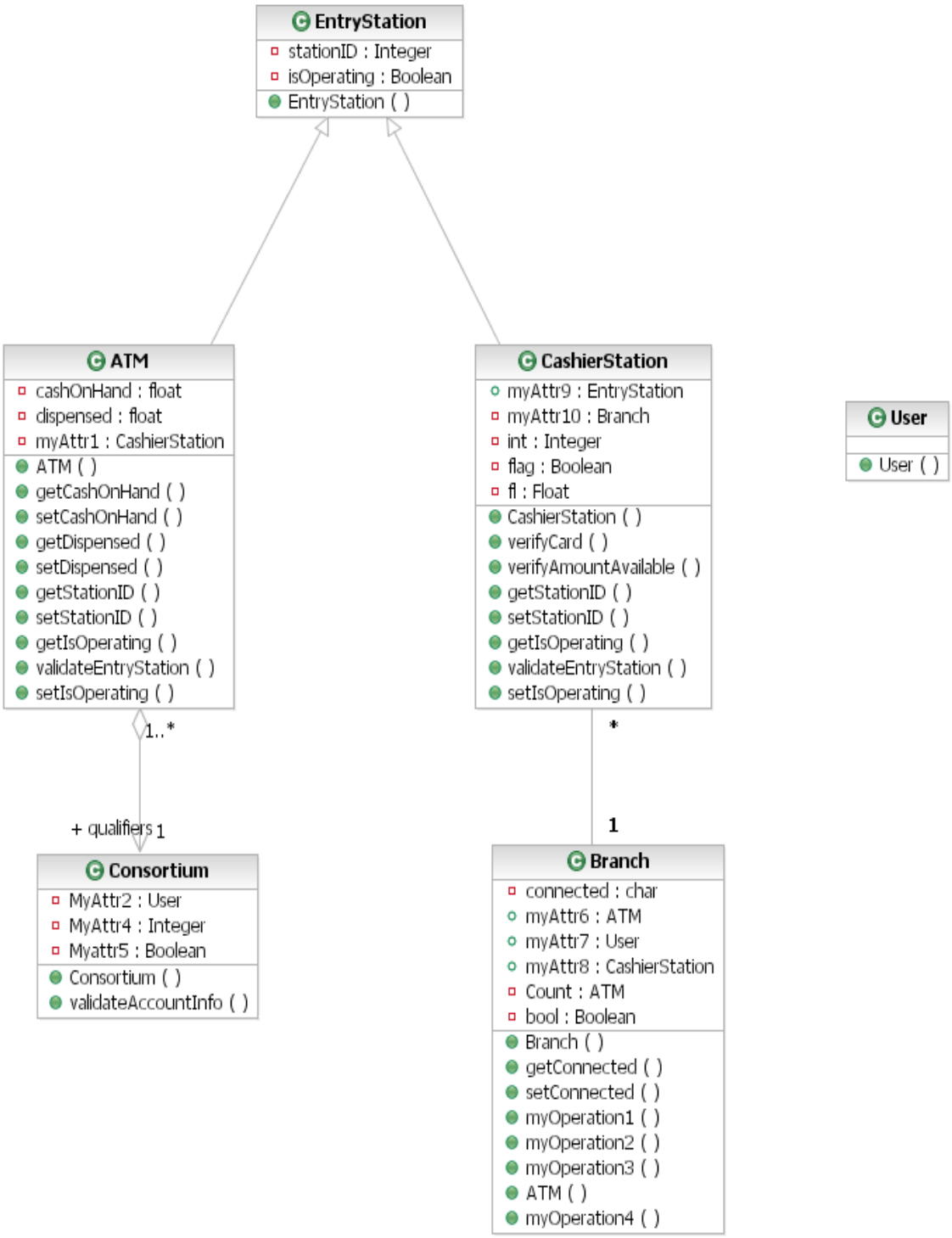


Figure 4.3: The Experimental Refined Class Diagram (with no "virtual" methods)

Design Metric	Value
DSC (Design Size)	6
NOH (Hierarchies)	1
ANA (Abstraction)	0.33
DAM (Encapsulation)	0.72
DCC (Coupling)	1.33
CAM (Cohesion)	2.50
MOA (Composition)	1.17
MFA (Inheritance)	0.03
NOP (Polymorphism)	0
CIS (Messaging)	5
NOM (Complexity)	5

Table 4.5: Design Metrics Values for the class diagram without "virtual" methods

Table 4.6a shows the values of the resulting quality attributes for the class diagram with no "virtual" methods while in table 4.6b we show the weighted average obtained for the same 3 cases of priority settings for the quality attributes.

Quality Attribute	Value
Reusability	4.54
Flexibility	0.43
Understandability	-4.77
Functionality	2.34
Extendibility	-0.48
Effectiveness	0.45

Table 4.6a: Quality attribute values for the class diagram without "virtual" methods

Case #	Weighted Average Value
Case I	0.42
Case II	2.87
Case III	0.47

Table 4.6b: The weighted averages obtained for the 3 cases of priority settings

The analysis report generated by SDAnalysis on the refined class diagram in figure 4.3 included one observation as follows:

Observation No. 1:

ATM and CashierStation are descendent classes; however, their own methods by far outnumber what they inherit from their parent classes. This negatively affects the extendibility and effectiveness quality attributes.

Discussion of Results

Examining the analysis report generated for the experimental version of our class diagram (in figure 4.3), we note that there is only one observation generated. This in turn supports our error analysis of the inheritance design property. However, as we compare the values of the quality attributes in table 4.6a with those in table 4.4a, we notice a significant drop in most quality attribute values which depend on inheritance. The change applied to the class diagram was intended to decrease the values of NOP and MFA, but unfortunately other design metrics (namely, CIS and NOM) were also greatly affected. This lead to the drop in the values of most of the quality attributes (reusability, functionality, flexibility, extendibility, and effectiveness). Hence we note here that although a design metric or two appear to have a high value (such as NOP and MFA) by themselves, altering their values can affect other design metrics. This experiment led us to believe that our judgments should not be restricted to the generated observations, but should be based on all 4 elements of the generated report, namely: the metrics' values, the quality attributes' values, the weighted average value, and finally the observation report. Our final goal is to identify the class diagram that has the best results in all 4 elements of the generated report.

Table 4.7 shows the comparison between values obtained for the 3 class diagrams (in figures 4.1, 4.2 and 4.3 above).

	Original Design Figure 4.1	**Modified Design** Figure 4.2	**Experimental Design** Figure 4.3
DSC	6	6	6
NOH	1	1	1
ANA	0.33	0.33	0.33
DAM	0.5	0.72 (↑ 0.22)	0.72
DCC	2	1.33 (↓ 0.67)	1.33
CAM	3.17	2.83 (↓ 0.34)	2.50 (↓ 0.33)
MOA	1.5	1.17	1.17
MFA	0.13	0.13	0.03 (↓ 0.10)
NOP	1.67	1.67	0
CIS	4	5.83 (↑ 1.83)	5 (↓ 0.83)
NOM	5.83	5.83	5 (↓ 0.83)
Reusability	3.71	4.88 (↑ 1.17)	4.54 (↓ 0.34)
Flexibility	1.21	1.27 (↑ 0.06)	0.43 (↓ 0.84)
Understandability	-6.10	-5.70 (↑ 0.40)	-4.88 (↑ 0.82)
Functionality	2.41	2.85 (↑ 0.44)	2.34 (↓ 0.51)
Extendibility	0.06	0.40 (↑ 0.34)	-0.48 (↓ 0.88)
Effectiveness	0.83	0.80	0.45 (↓ 0.35)
Weighted Average (Case I)	0.35	0.75	0.42
Weighted Average (Case II)	2.58	3.33	2.87
Weighted Average (Case III)	0.66	1.20	0.47

Table 4.7: Comparison of design metrics and quality attributes for all 3 class diagrams

As we compare between the quality attribute values obtained for the initial class diagram (in table 4.2a) with those obtained after the modifications (in table 4.4a), we observe the following:

1) The reusability and extendibility quality attribute values increased noticeably. The design metric which is common between these two quality attributes is the Direct Class Coupling (DCC). If we consider the changes made to the class diagram which affected the DCC from table 4.1 and table 4.3, we find that the value of the DCC has dropped from 2 to 1.33. This drop in the DCC value was automatically reflected on both the reusability and extendibility quality attributes.

2) There are two other design metrics that affected the value of the reusability quality attribute, which are the Class Interface Size (CIS) and the Cohesion Among Methods in Class (CAM). These metrics also affected the value of the functionality quality attribute. The CIS value improved from 4 to 5.83 which means that the number of public methods increased in the overall design. The CAM value decreased from 3.17 to 2.83 leading to improved functionality and reusability.

3) The flexibility and understandability quality attributes values increased slightly. The design metrics which affect these two quality attributes are the Data Access Metric (DAM), Number of Polymorphic Methods (NOP), and DCC. Closer inspection shows that the little increase in the quality attributes came from the DAM as its value increased from 0.5 to 0.72. This increase was not clear enough as it was dominated by the decrease which occurred in the DCC value from 2 to 1.33. As the value for the NOP did not change, we gather that it did not affect the values of the flexibility and understandability quality attributes.

If we compare the calculated weighted average for figure 4.1 with that for figure 4.2, we notice that in the three cases of priority settings, the values for the calculated weights improved (from 0.35 to 0.75 in case I, from 2.58 to 3.33 in case II, and from 0.66 to 1.20 in case III). However, if we compare the values for figure 4.2 and figure 4.3 (as shown in

Table 4.7; Modified Design and Experimental Design), the drop in all the quality attributes negatively affected the results of the calculated weighted average (from 0.75 to 0.42 in case I, from 3.33 to 2.87 in case II, and from 1.20 to 0.47 in case III). It is also important to clarify that the relatively large values of weighted average in case II are due to the Reusability quality attribute whose values are much higher compared to all the other quality attributes.

From all the computed metrics, quality attributes, and weighted average values listed in Table 4.7, we are assured that figure 4.2 is the most appropriate design for this example as it gave the most acceptable quality attribute values and weighted average. Figure 4.1 had serious weaknesses which were fixed in figure 4.2 whereas figure 4.3 (in which we experimented by minimizing the NOP), the results got worse because the inheritance became meaningless and in turn had an adverse effect on the quality attributes that depend on MFA (Measure of Functional Abstraction, the Inheritance measure).

As a final observation we note that most quality attributes decreased as indicated in the brackets in Table 4.7, with figure 4.3. This is clearly attributed to the drop in most of the design metrics in the same figure while most metrics (and quality attributes) increased, except for the DCC and CAM values, with figure 4.2.

4.2 The Second Example

In this section, we present our second example of an adapted class diagram [23] in figure 4.4 with a different set of weak design attributes and we show how our analysis report helped in correcting this class diagram.

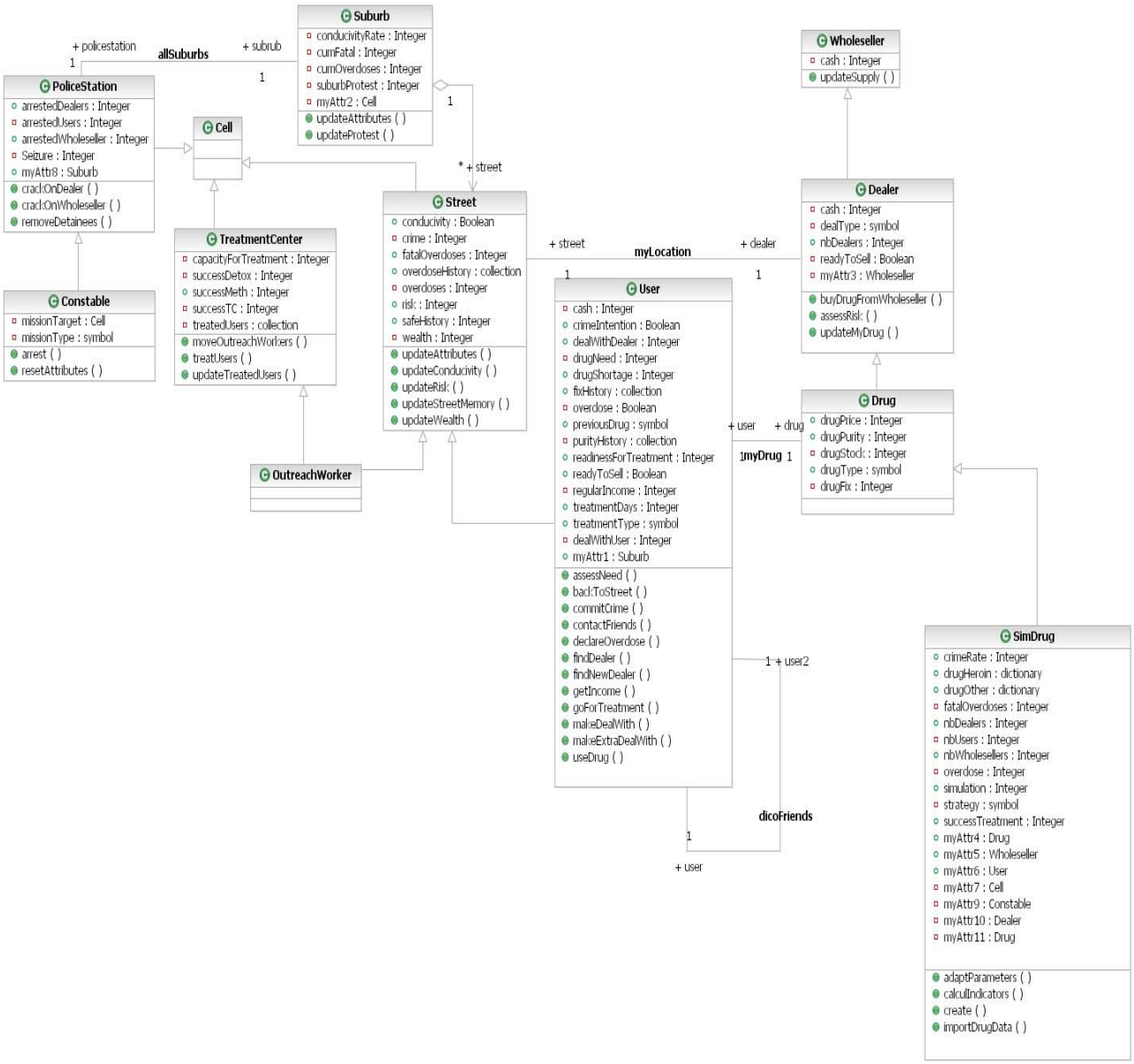


Figure 4.4: The Second Adapted Class Diagram [23]

We applied 3 cases of priority settings on this example. In the first case (**case I**) we applied an equal priority scheme for the quality attributes: Reusability, Functionality, Flexibility, Extendibility, Effectiveness and Understandability.

In the second case (**case II**) we assigned each quality attribute a separate priority. The priority setting in this case is as follows:

Priority 1 : Effectiveness

Priority 2 : Functionality

Priority 3 : Understandability

Priority 4 : Flexibility

Priority 5 : Extendibility

Priority 6 : Reusability

In the third case **(case III)** we gave two quality attributes higher priorities and the remaining attributes were given equal (third) priority. The priorities were set as follows:

Priority 1 : Flexibility

Priority 2 : Extendibility

Priority 3 : Reusability, Effectiveness, Understandability, Functionality

Table 4.8 shows the set of design metrics and their values as computed for the class diagram in figure 4.4. These values were calculated by using both the SDMetrics and SDAnalysis tools.

Design Metric	**Value**
DSC (Design Size)	12
NOH (Hierarchies)	2
ANA (Abstraction)	1.33
DAM (Encapsulation)	0.55
DCC (Coupling)	0.92
CAM (Cohesion)	3.25
MOA (Composition)	0.92
MFA (Inheritance)	0.30
NOP (Polymorphism)	0
CIS (Messaging)	2.92
NOM (Complexity)	2.92

Table 4.8: The values for the design metrics for the adapted class diagram in figure 4.4

The SDAnalysis tool calculated the quality attributes according to the formulas set by Bansiya et al [2]. Table 4.9a shows the values of these quality attributes for the class diagram in figure 4.4 and Table 4.9b shows the weights obtained for the 3 cases of priority settings.

Quality Attribute	Value
Reusability	6.42
Flexibility	0.37
Understandability	-6.56
Functionality	3.33
Extendibility	0.36
Effectiveness	0.62

Table 4.9a: The quality attributes values for the adapted class diagram in figure 4.4

Case #	Weighted Average Value
Case I	0.76
Case II	0.49
Case III	0.85

Table 4.9b: The weighted averages obtained for the 3 cases of priority settings

The analysis report generated by the SDAnalysis tool for the class diagram in figure 4.4 included two observations which are as follows:

Observation No. 1:

The Street, User, and SimDrug classes have a larger number of public attributes than private attributes and this is structurally unfavorable. Their encapsulation values are low and therefore, they can affect the overall values of flexibility, effectiveness, and understandability quality attributes.

Observation No. 2:

The PoliceStation, TreatmentCenter, Street, Dealer, and User are descendent classes; however, their own methods by far outnumber what they inherit from their parent classes. This negatively affects the extendibility and effectiveness quality attributes.

Each class was revised separately and the defects detected and repaired as follows:

1. The attributes in the SimDrug class were examined and it was noted that they are not used outside the class and hence, the visibility of all attributes was changed from public to private.
2. Similarly the attributes in the User class were revised and their visibility changed from public to private. Also, upon a close inspection, we noted that the relationship between the User class and the Street class could be an association relationship instead of inheritance.
3. Equally the relationship between the Wholeseller class and the Dealer class need not to be an inheritance relationship. A simple association relationship could be more applicable.
4. The PoliceStation, TreatmentCenter, and Street are inherited classes from the Cell class. However, we noted that the Cell class contains no attributes or methods. We made each of these classes a stand alone class and removed the inheritance relationship between them and the Cell class.

The above solutions were implemented based on the analysis report. After repairing the identified errors, the class diagram was restructured. The new corrected diagram is shown in figure 4.5.

Table 4.10 shows all the design metric values extracted from the class diagram by SDMetrics and calculated by the SDAnalysis tool.

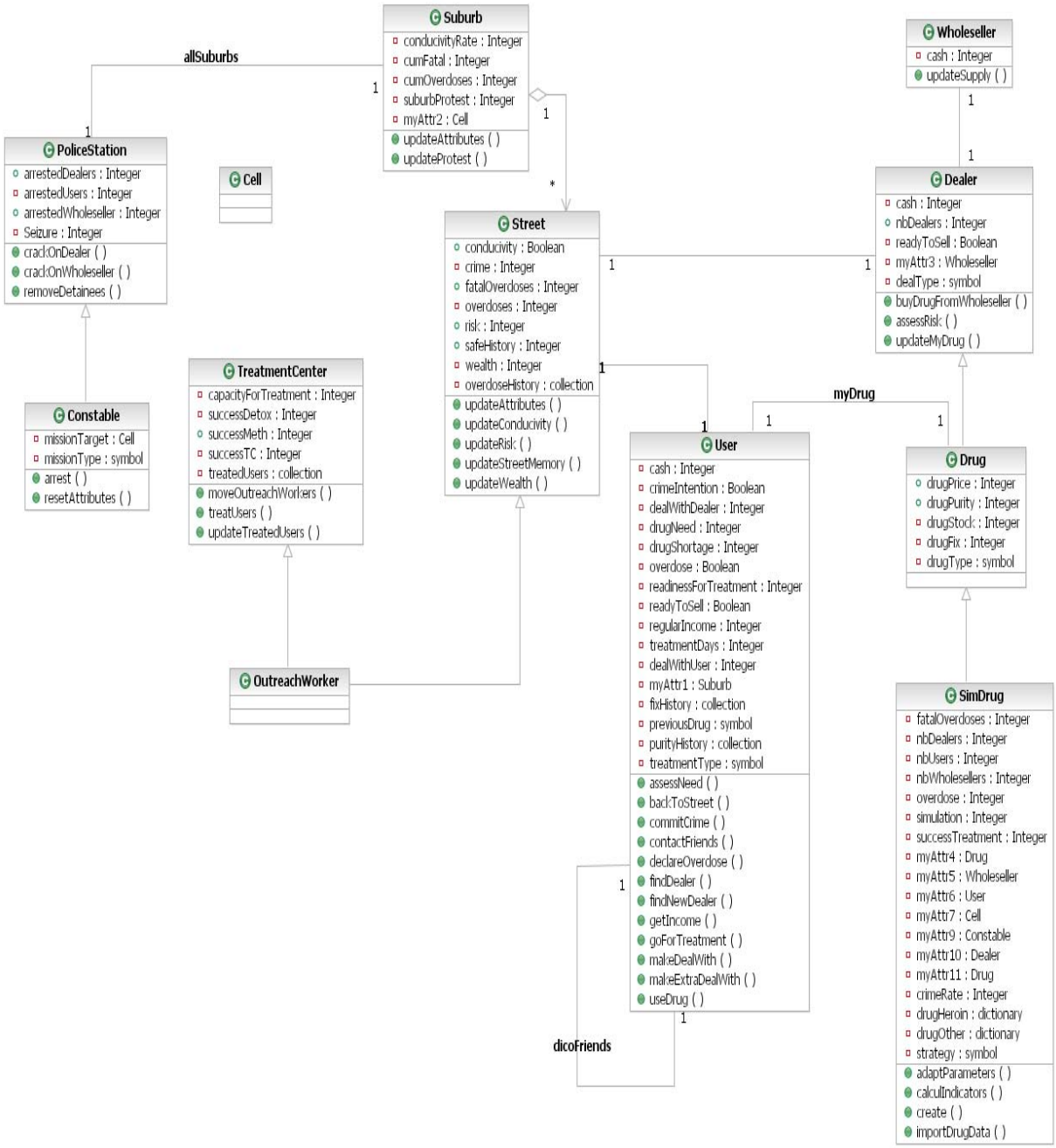


Figure 4.5: The Refined Class Diagram (for the second adapted example)

Table 4.11a shows the corresponding values for the resulting quality attributes for the refined class diagram. Table 4.11b shows the weights obtained for the 3 cases of priority settings.

Design Metric	Value
DSC (Design Size)	12
NOH (Hierarchies)	4
ANA (Abstraction)	0.5 (0.33)
DAM (Encapsulation)	0.68
DCC (Coupling)	0.83
CAM (Cohesion)	2.25 (2.17)
MOA (Composition)	0.83
MFA (Inheritance)	0.25 (0.22)
NOP (Polymorphism)	0
CIS (Messaging)	2.92
NOM (Complexity)	2.92

Table 4.10: Design metric values for the refined class diagram in figure 4.5 (and figure 4.6)

Quality Attribute	Value
Reusability	6.69 (6.71)
Flexibility	0.38
Understandability	-5.88 (-5.80)
Functionality	3.89 (3.90)
Extendibility	-0.04 (-0.14)
Effectiveness	0.45 (0.41)

Table 4.11a: Quality attribute values for the refined class diagram in figure 4.5 (and figure 4.6)

Case #	Weighted Average Value
Case I	0.92 (0.91)
Case II	0.63 (0.62)
Case III	0.93 (0.91)

Table 4.11b: The weighted averages obtained for the 3 cases of priority settings for the class diagram in figure 4.5 (and figure 4.6)

Only one observation was generated by SDAnalysis on the refined class diagram in figure 4.5 which is as follows:

Observation No. 1:

SimDrug is a descendent class; however, its own methods by far outnumber what it inherits from its parent class. This negatively affects the extendibility and effectiveness quality attributes.

According to the above report the SimDrug class was further examined and we noted that we could do away with the inheritance relationship between this class and the Drug class. The changes are shown in figure 4.6. Deleting this relationship from the model had a positive impact on the ANA and CAM design metrics, but a negative effect on the MFA design metrics. Their new values are indicated in brackets in table 4.10, next to the original values. The values of the quality attributes that were affected by the changes in the design metrics are indicated in brackets in table 4.11a and their corresponding weighted averages are indicated in brackets in table 4.11b. Only the Extendibility attribute got negatively affected while the remaining attributes remained almost the same. The decrease in the Extendibility attribute (from -0.04 to -0.14) was a result of the decrease in the ANA (from 0.5 to 0.33) and the slight decrease in the MFA (from 0.25 to 0.22).

The improvements in the refined class diagram (in figure 4.6) may be attributed to the following:

- A slight enhancement occurred in two quality metrics while the rest of the metrics remained unchanged.
- Most of the quality attributes were slightly (positively) affected except for the Extendibility, which contains both ANA and MFA as metrics in its equation, and they both decreased in value when compared to the calculated metrics in figure 4.5.
- There is no reported observation on this class diagram (figure 4.6).
- The calculated weighted averages in the three cases for both figure 4.5 and 4.6 are very similar.

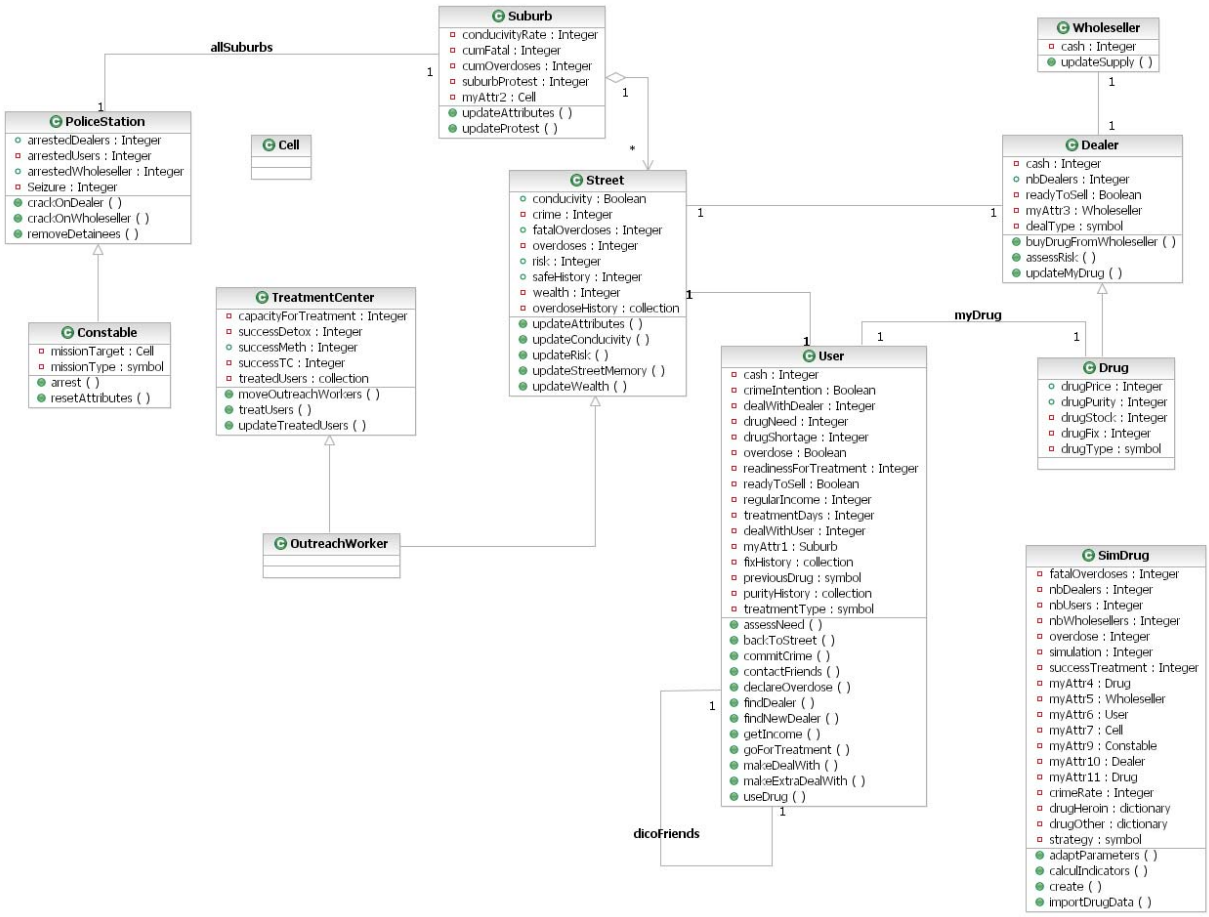


Figure 4.6: The Refined Class Diagram without SimDrug/Drug Inheritance

However, for the sake of experimentation, we wished to test the effect of changing the inheritance relationships between classes, when it is not needed, to simple association. We returned to the original diagram in figure 4.4 where there were inheritance relationships between the PoliceStation, TreatmentCenter, and Street classes with the Cell class. The change that we suggested was to group common attributes and common methods from all 3 classes and define them in the Cell class. The detailed changes made were as follows:

1. Setting the visibility of all attributes to private.
2. Finding similar attributes and methods in PoliceStation, TreatmentCenter, and Street classes, deleting them from these classes, and declaring them in the Cell class. Then re-establishing the inheritance relationships between each of the three classes and the Cell class.

3. Changing the relationship between the Dealer and Drug classes from inheritance to association.
4. Changing the relationship between the OutreachWorker class and TreatmentCenter and Street classes from inheritance to association.
5. Changing the relationship between the Constable and PoliceStation classes from inheritance to association.
6. Checking all classes for excess usage of attributes which had resulted in that the SimDrug class has two attributes (myAttr4 and myAttr11) which are of the same type (class type: Drug). Therefore, we deleted the attribute myAttr11.

Figure 4.7 shows the new class diagram after the above changes were applied.

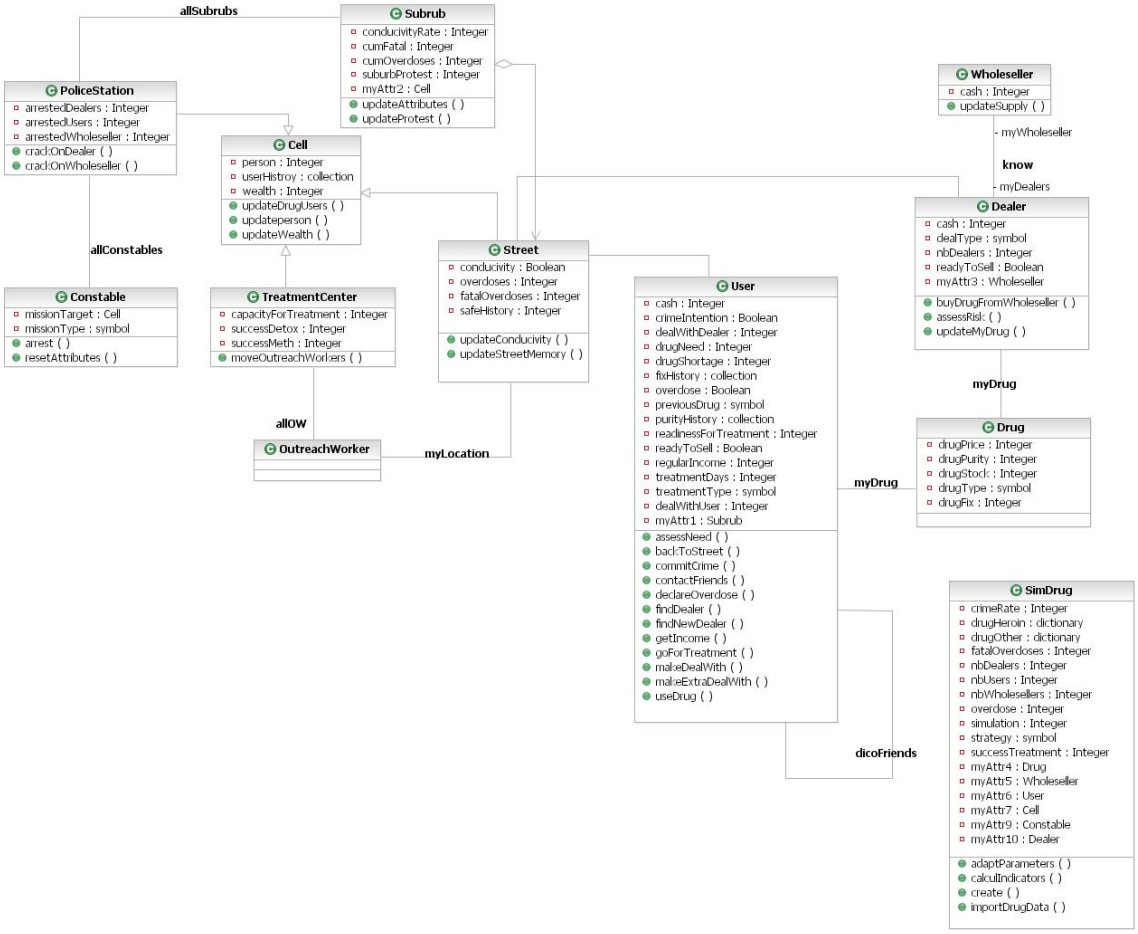


Figure 4.7: Applying New Changes to the Class Diagram in figure 4.4 (Experimental)

Table 4.12 shows all the values extracted from the class diagram by SDMetrics and calculated by the SDAnalysis tool.

Design Metric	Value
DSC (Design Size)	12
NOH (Hierarchies)	1
ANA (Abstraction)	0.25
DAM (Encapsulation)	0.92
DCC (Coupling)	0.83
CAM (Cohesion)	2.75
MOA (Composition)	0.83
MFA (Inheritance)	0.16
NOP (Polymorphism)	0
CIS (Messaging)	2.67
NOM (Complexity)	2.67

Table 4.12: Design metrics values for the new changed class diagram

Table 4.13a shows the values for the resulting quality attributes for the new class diagram after implementing the above changes. Table 4.13b shows the weights obtained for the 3 cases of priority settings.

Quality Attribute	Value
Reusability	6.44
Flexibility	0.44
Understandability	-5.80
Functionality	3.12
Extendibility	-0.21
Effectiveness	0.43

Table 4.13a: Quality attribute values for the new changed class diagram

Case #	Weighted Average Value
Case I	0.74
Case II	0.43
Case III	0.78

Table 4.13b: The weighted averages obtained for the 3 cases of priority settings

The final report did not include any observations.

Discussion of Results

As we examine the experimental version of our class diagram, we note that there is no observation generated. This in turn supports our experimental hypotheses in which we claim that 1) changing the inheritance relationships between classes, when it is not needed, to simple association and 2) using the inheritance relationship when we find more than one class having similar attributes and similar methods, leads to better metric values.

Table 4.14 compares all the design metrics and quality attribute values extracted from the above 4 class diagrams.

	Original Design (fig. 4.4)	**Refined Class Diagram (fig. 4.5)**	**Without SimDrug/Drug Inheritance (fig. 4.6)**	**Experimental Design (fig. 4.7)**
DSC	12	12	12	12
NOH	2	4	4 (↑ 2)	1 (↓ 3)
ANA	1.33	0.5	0.33 (↓ 1)	0.25 (↓ 0.08)
DAM	0.55	0.68	0.68 (↑ 0.13)	0.92 (↑ 0.24)
DCC	0.92	0.83	0.83 (↓ 0.09)	0.83
CAM	3.25	2.25	2.17 (↓ 1.08)	2.75 (↑ 0.58)
MOA	0.92	0.83	0.83	0.83
MFA	0.30	0.25	0.22 (↓ 0.08)	0.16 (↓ 0.06)
NOP	0	0	0	0
CIS	2.92	2.92	2.92	2.67 (↓ 0.25)
NOM	2.92	2.92	2.92	2.67 (↓ 0.25)
Reusability	6.42	6.69	6.71 (↑ 0.29)	6.44 (↓ 0.27)
Flexibility	0.37	0.38	0.38	0.44 (↑ 0.06)
Understandability	-6.56	-5.88	-5.80	-5.80
Functionality	3.33	3.89	3.90 (↑ 0.57)	3.12 (↓ 0.78)
Extendibility	0.36	-0.04	-0.14 (↓ 0.5)	-0.21 (↓ 0.07)
Effectiveness	0.62	0.45	0.41 (↓ 0.21)	0.43 (↑ 0.02)
Weighted Average (Case I)	0.76	0.92	0.91 (↑ 0.15)	0.74 (↓ 0.17)
Weighted Average (Case II)	0.49	0.63	0.62 (↑ 0.13)	0.43 (↓ 0.19)
Weighted Average (Case III)	0.85	0.93	0.91 (↑ 0.06)	0.78 (↓ 0.13)

Table 4.14: Comparison of design metrics and quality attributes for all 4 class diagrams in figures 4.4 through 4.7

Figures 4.6 and 4.7 did not generate any observations and we claim that they provide the best solutions. Figure 4.5 resulted in one observation that was resolved in figure 4.6. Therefore, we focus on figures 4.6 and 4.7 class diagrams and discuss their results. Nevertheless, we still need to examine the rest of the analysis report to support our claim. If we compare the quality attributes and hence the design metrics of figure 4.6 with those obtained for figure 4.5, we note the following:

1) Figure 4.6 shows an increase in Reusability which is attributed to the sharp drop in CAM by 1.08.
2) Functionality is another quality attribute that exhibited a fair increase in its value (by 0.57). The most distinguishable indicator in this quality attribute is NOH (as it is not found in any other quality attributes' equation) whose value increased by 2. Also, the decrease that occurred in the CAM was reflected on the increase in Functionality.
3) However figure 4.6 shows a drop in both Extendibility and Effectiveness quality attributes. The common design metric in both equations is ANA and its value decreased from 1.33 to 0.33. However, the decrease in Effectiveness was not as high as that in Extendibility as it was dominated by the increase that occurred in the DAM.
4) Finally, figure 4.6 shows an interesting increase in the weighted average value in all 3 priority cases in. This supports our first judgment that figure 4.6 could be considered one of the best solutions for the original problem.

As we compare the values of the quality attributes in the Experimental class diagram (figure 4.7) with those in the Refined (Without SimDrug/Drug Inheritance) class diagram (figure 4.6), we notice the following:

1) Figure 4.7 shows an opposite result to what was calculated for figure 4.6 where both Reusability and Functionality decreased. Apparently, the decrease in Reusability came from the increase in CAM where we increased the number of relationships between classes when we restored the inheritance relationships in figure 4.4.

2) Functionality decreased tremendously as the number of hierarchies (NOH) dropped from 4 to 1. Also, the decrease in messaging (CIS) contributed to this decrease where it had dropped by 0.25.
3) The last decrease that occurred to a quality attribute was by a very minor proportion from that of figure 4.6 where Extendibility decreased by 0.07.
4) Finally, figure 4.7 shows a high drop in the values of the weighted average for the 3 priority cases.

Although figure 4.7 shows lower values in a number of metrics, quality attributes, and weighted averages when compared to the values in figure 4.6, this could be attributed to the retained inheritance relationships between the Cell class and the PoliceStation, TreatmentCenter, and Street classes.

4.3 The Third Example

In this section, we present our third example of an adapted class diagram [24] with a different set of weak design attributes and we show how our metrics/attributes analysis technique can lead us to refine this class diagram. The initial class diagram is shown in Figure 4.8.

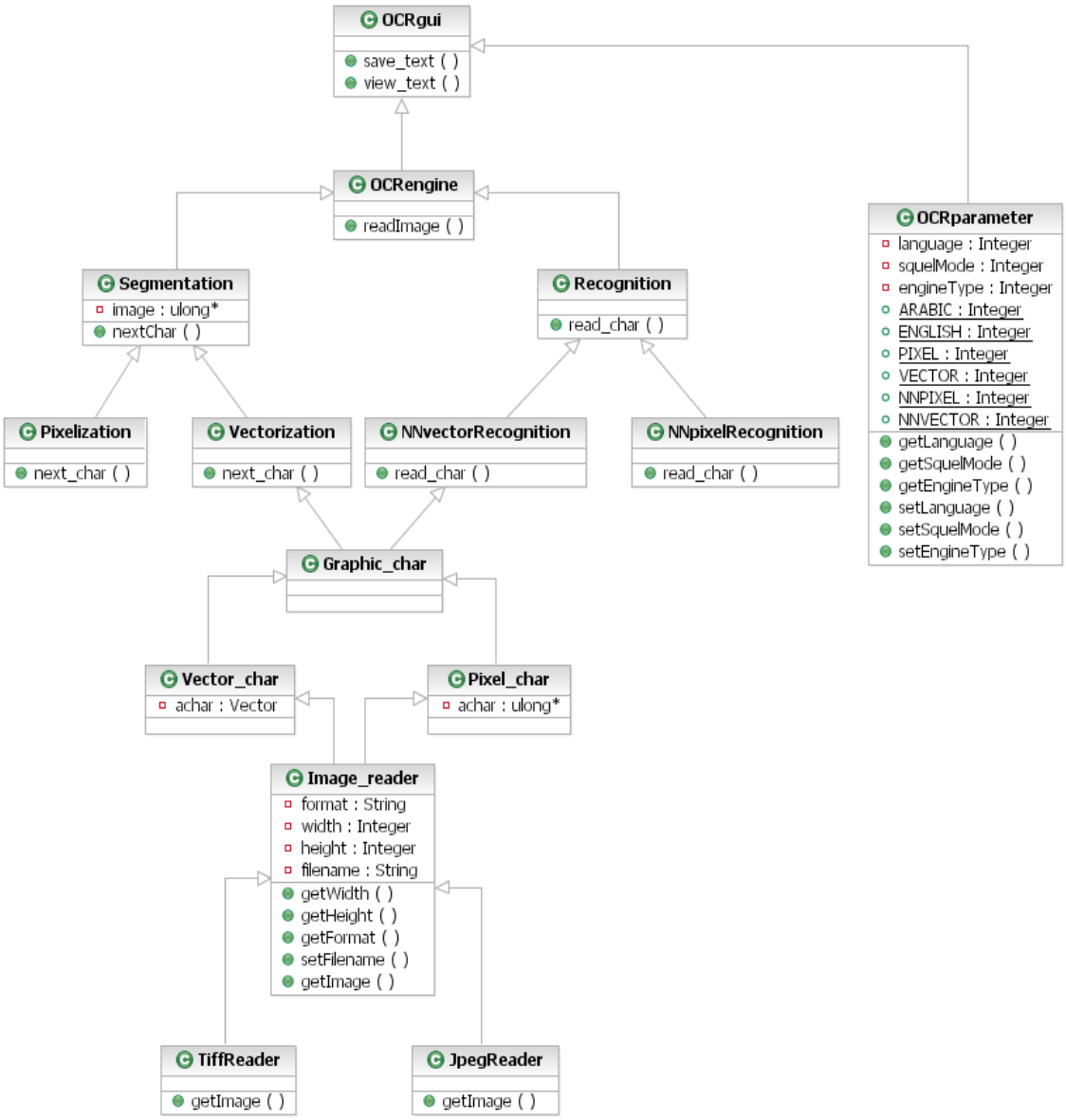


Figure 4.8: Third Adapted Class Diagram [24]

As in the previous 2 examples, we applied 3 cases of priority settings on this example. In the first case **(case I)** we applied an equal priority scheme for the quality attributes: Reusability, Functionality, Flexibility, Extendibility, Effectiveness, and Understandability.

In the second case **(case II)** we assigned each quality attribute a separate priority as follows:

Priority 1 : Understandability

Priority 2 : Reusability

Priority 3 : Functionality

Priority 4 : Flexibility

Priority 5 : Effectiveness

Priority 6 : Extendibility

In the third case **(case III)** we gave two quality attributes higher priorities and the remaining attributes were given equal (third) priority. The priorities were set as follows:

Priority 1 : Effectiveness

Priority 2 : Extendibility

Priority 3 : Reusability, Flexibility, Understandability, Functionality

Table 4.15 presents all the design metrics and their values as computed for the class diagram in figure 4.8. These values were calculated by using both the SDMetrics and SDAnalysis tools.

Design Metric	**Value**
DSC (Design Size)	15
NOH (Hierarchies)	1
ANA (Abstraction)	4.47
DAM (Encapsulation)	0.29
DCC (Coupling)	0
CAM (Cohesion)	1.13
MOA (Composition)	0
MFA (Inheritance)	0.74
NOP (Polymorphism)	0
CIS (Messaging)	1.47
NOM (Complexity)	1.47

Table 4.15: The Design Metrics for the Adapted Class Diagram in figure 4.8

Table 4.16a shows the values of these quality attributes for the class diagram in figure 4.8 and Table 4.16b shows the weights obtained for the 3 cases of priority settings.

Quality Attribute	Value
Reusability	7.95
Flexibility	0.07
Understandability	-7.19
Functionality	3.71
Extendibility	2.61
Effectiveness	1.1

Table 4.16a: The quality attributes values for the adapted class diagram

Case #	Weighted Average Value
Case I	1.37
Case II	-1.03
Case III	1.99

Table 4.16b: The weighted averages obtained for the 3 cases of priority settings

The analysis report generated by the SDAnalysis tool for the class diagram in figure 4.8 included a number of observations which are as follows:

Observation No. 1:

The design consists of one hierarchy structure and there is high dependency between classes. This negatively affects the functionality quality attribute of the whole design.

Observation No. 2:

The design consists of a huge hierarchy structure which means that there is no abstraction. This in turn affects the extendibility and effectiveness of the overall design.

Observation No. 3:

The OCRparameter class has more public attributes than private attributes and this is structurally unfavorable. Its encapsulation value is low and therefore, it can affect the overall values of flexibility, effectiveness, and understandability quality attributes.

Observation No. 4:

The OCRparameter is a descendent class; however, its own methods by far outnumber what it inherits from the parent class. This negatively affects the extendibility and effectiveness quality attributes.

Each class was revised separately and the defects detected and repaired as follows:

1. The inherited relationships that connect to the OCRgui were replaced by simple association since OCRengine and OCRparameter do not inherit any of the functions in OCRgui.
2. Image_reader and Graphic_char classes were extracted from the long hierarchy tree and were placed as two separate classes with their own subclasses.
3. The OCRparameter class was examined separately and the following was noticed:
    a. Observation No. 4 is automatically solved after changing the relationship between this class and the OCRgui to be a simple association (as resolved in 1 above).
    b. The public attributes found in the class are constant attributes and they need to be public to be accessible from any other class.

The above solutions were implemented based on the analysis report. After repairing the identified errors that were detected in the observations, the class diagram was restructured. The new corrected diagram is shown in Figure 4.9.

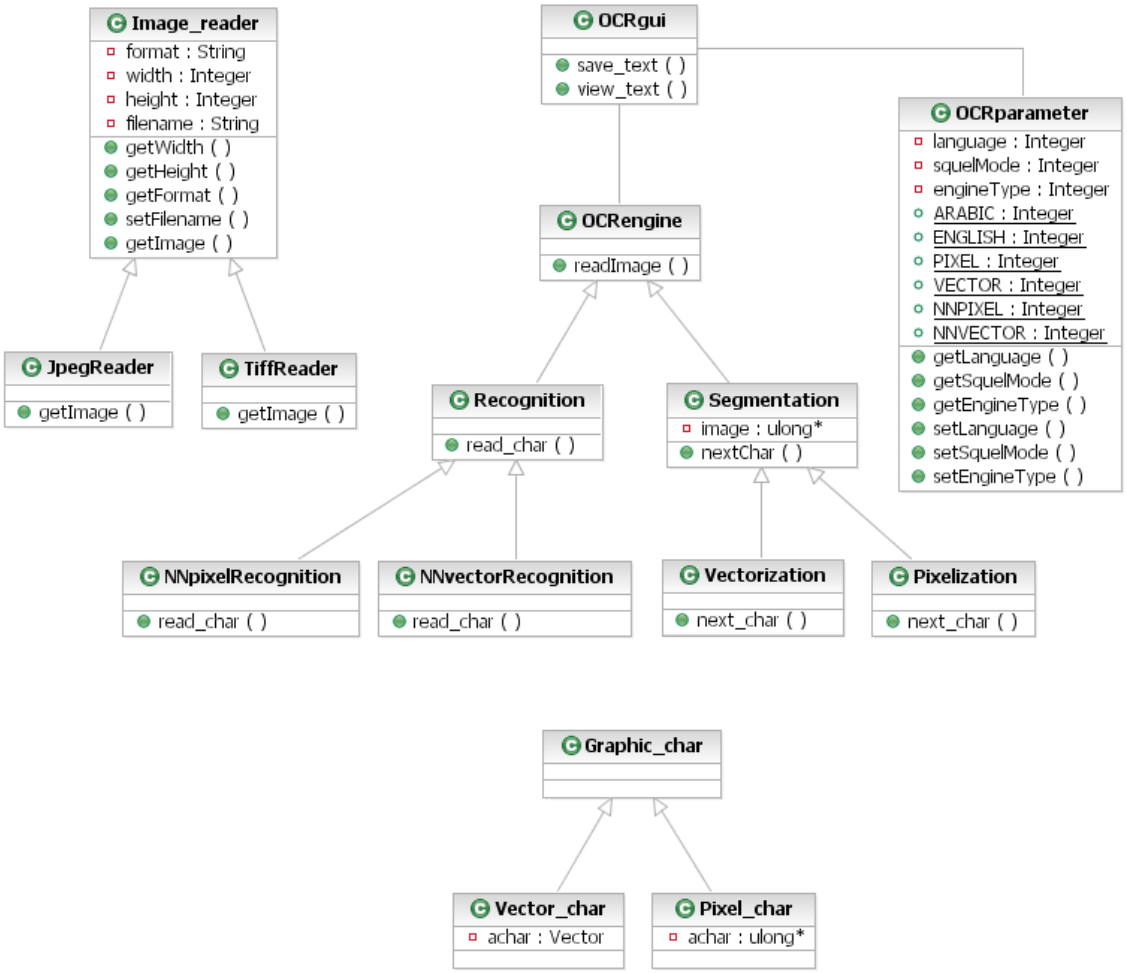


Figure 4.9: The Refined Class Diagram (third example)

Table 4.17 shows the computed design metric values for the refined class diagram in figure 4.9.

Design Metric	Value
DSC (Design Size)	15
NOH (Hierarchies)	3
ANA (Abstraction)	0.93
DAM (Encapsulation)	0.29
DCC (Coupling)	0
CAM (Cohesion)	1
MOA (Composition)	0
MFA (Inheritance)	0.36
NOP (Polymorphism)	0
CIS (Messaging)	1.47
NOM (Complexity)	1.47

Table 4.17: Design Metric Values for the Refined Class Diagram in Figure 4.9

Table 4.18a shows the values for the resulting quality attributes for the refined class diagram. Table 4.18b shows the weights obtained for the 3 cases of priority settings.

Quality Attribute	Value
Reusability	7.98
Flexibility	0.07
Understandability	-5.98
Functionality	4.16
Extendibility	0.65
Effectiveness	0.32

Table 4.18a: Quality attribute values for the refined class diagram in figure 4.9

Case #	Weighted Average Value
Case I	1.20
Case II	-0.41
Case III	1.23

Table 4.18b: The weighted averages obtained for the 3 cases of priority settings

Only one observation was generated by SDanalysis on the refined class diagram in figure 4.9 which is as follows:

Observation No. 1:

The OCRparameter class has more public attributes than private attributes and this is structurally unfavorable. Its encapsulation value is low and therefore, it can affect the overall values of flexibility, effectiveness, and understandability quality attributes.

The public attributes are constants that could be used by any other class; and therefore, we could not work any further on this observation.

Discussion of Results

We selected the class diagram in figure 4.8 as it illustrates the typical problems that result from a dense hierarchy structure. As noted in figure 4.8, the class diagram started with seven levels of inheritance. The most significant design properties here are hierarchies and abstraction. The corresponding design metrics are Number of Hierarchies (NOH) and Average Number of Ancestors (ANA) respectively. According to the total number of classes in the diagram, if the depth of inheritance exceeds half the worst value for ANA (which is △DSC/DSC), then this will be expected to negatively affect understandability, extendibility, effectiveness and functionality quality attributes. This is easily shown in the calculated data in Tables 15 and 16a.

Table 4.19 presents a comparison between the design metrics, the quality attributes, and the weighted average values obtained for the initial class diagram (in figure 4.8) with those obtained for the refined diagram (in figure 4.9).

	Original Design Figure 4.8	**Modified Design** Figure 4.9
DSC	15	15
NOH	1	3 (↑ 2)
ANA	4.47	0.93 (↓ 3.54)
DAM	0.29	0.29
DCC	0	0
CAM	1.13	1 (↓ 0.13)
MOA	0	0
MFA	0.74	0.36 (↓ 0.38)
NOP	0	0
CIS	1.47	1.47
NOM	1.47	1.47
Reusability	7.95	7.98 (↑ 0.03)
Flexibility	0.07	0.07
Understandability	-7.19	-5.98 (↑ 1.21)
Functionality	3.71	4.16 (↑ 0.45)
Extendibility	2.61	0.65 (↓ 1.96)
Effectiveness	1.1	0.32 (↓ 0.78)
Weighted Average (Case I)	1.37	1.20 (↓ 0.17)
Weighted Average (Case II)	-1.03	-0.41 (↑ 0.62)
Weighted Average (Case III)	1.99	1.23 (↓ 0.76)

Table 4.19: Comparison between the Design Metrics, Quality Attributes, and Weighted Averages for the class diagrams in figures 4.8 and 4.9

As we compare the quality attribute values for the 2 class diagrams in Table 4.19, we note the following:

1) The functionality quality attribute's value increased due to the increase in the NOH (from 1 to 3). The NOH design metric is only found in the functionality quality attribute equation.
2) All the quality attributes that have either MFA (Measure of Functional Abstraction) or ANA as terms in their equations decreased in value. This fact is quite noticeable in the extendibility and effectiveness quality attributes' values as they both have the MFA as well as the ANA in their equations. It is quite noticeable that the drop in the extendibility is more than that of the effectiveness due to the fact that the coefficient in the extendibility equation (0.5) is greater than that of the effectiveness (0.2).
3) The value for understandability increased due to the decrease encountered in the design metric ANA.

If we compare the weighted average values for the 3 priority cases (I, II and III) for the 2 class diagrams (in figure 4.8 and figure 4.9), we note the following:

1) In Case I, where we have equal weights for all quality attributes, there is a net decrease in the weighted average since we have a decrease in the values of two quality attributes (extendibility and effectiveness) with a total drop of 2.74, while the other three quality attributes (reusability, understandability, and functionality) increased by 1.69.
2) In Case II, where understandability gets the highest weight and extendibility the lowest weight, the weight on the increase in understandability is higher than the weight on the decrease in extendibility. This in turn, resulted in a net increase in the weighted average.
3) In Case III, we wanted to go in the opposite direction of Case II. The two quality attributes (effectiveness and extendibility) whose values dropped down significantly were given the first and second priorities and the remaining quality attributes were given equal priority. The net result on the weighted average value was a sharper decrease.

We experimented with a part of the MFC version 7.0 class diagram [9] and our results showed the consistency with Bansiya et al's results [2] that if the number of classes by far out number the level of inheritance, then the class design is acceptable. We show the drawn part of the MFC and its corresponding NOH and ANA values in Appendix C.

Chapter 5: Analysis of Results

In this chapter, we highlight the significance of our findings. We sum up all our experimental results, which were based on relationships set by previous researchers between class design metrics and product quality metrics, and we try to draw fine lines through our findings. The chapter is divided into 4 main sections which represent the components of the analysis report (the output of our solution approach). The Design metrics are discussed in section 5.1, The Quality Attributes in section 5.2, the Weighted Average in section 5.3, and the Observations in section 5.4.

5.1 Design Metrics

From our experimental work with the design metric calculations recommended by previous researchers and complemented by our tool calculations, we got to see more clearly, the profound relationship between class design metrics and product quality attributes. We were able to identify which metrics should increase and which should decrease in order to improve the product quality attributes. Our objective was to improve the quality of the overall class diagram design which would lead to a better quality of the product. Through the metric calculations we are more confident in guiding the user improve the class diagram. In this section we discuss our findings about the effect of each metric, based on our experimental results.

5.1.1 Design Size in Classes – DSC

The importance of this metric is that it provides the basic figure on which the rest of the metrics depend. It is the denominator in most of our computations. We take the average of other metrics based on the total number of classes (DSC) and accordingly we evaluate the influence of these metrics positively or negatively with respect to the entire class diagram.

5.1.2 Average Number of Ancestors – ANA

It is acceptable to increase the value of the ANA as it indicates the average level of inheritance in the whole class diagram. However, if the value of ANA increases to more than 1/2(△DSC/DSC), i.e. if the number of ancestors increases to more than 1/2(△DSC/DSC), then this could lead to unfavorable results. Hence, in order for the value of ANA to be controllable within its optimization range, we suggested to set its acceptable range to be more than zero and less than 1/2(△DSC/DSC).

5.1.3 Number of Hierarchies – NOH

It is preferable to increase the number of hierarchies in a design than to have deeper levels of inheritance and to have every group of classes linked together in one bundle. It is important to avoid the two worst cases of NOH: the first where there is no inheritance as this means that there is no structure. The second is linked to ANA, where we should not have a deep level of inheritance that exceeds 1/2(△DSC/DSC).

5.1.4 Data Access Metric – DAM

The value of DAM should increase in order to elicit a positive effect on the class diagram under study. A large value of DAM implies that the classes possess a high encapsulation property. Our results support this argument. A larger value of DAM could be reached by minimizing the number of public attributes in the class diagram. Hence, we make sure to have public attributes only when they will be needed by other classes.

5.1.5 Direct Class Coupling – DCC

Our experimental results complied with what was stated in the literature about the negative effect of increased coupling between classes. It is thus preferable to keep the value of the design metric DCC as low as possible. This could be achieved by minimizing the number of class attributes or parameters in methods that are of other class types. We need to make sure that the classes in our class diagram are self dependent.

5.1.6 Cohesion Among Methods of Class – CAM

This was the most difficult measure to work with. We took its value directly from the SDMetrics tool [36]. However, the cohesion property in SDMetircs was computed differently from the commonly known value. According to the literature, it is better to increase the cohesion (the quantitative indication of the degree to which a module – in this case a class – focuses on just one thing [20]). SDMetrics defines cohesion as quantifying the connectivity between elements of the design unit: the higher the connectivity between elements the higher the cohesion. SDMetrics thus seeks lower values for CAM. This meant that it was better to minimize the interrelatedness between classes since the more the interrelatedness the less is the cohesion. In our experimental examples we sought to decrease the value of cohesion i.e. the interrelatedness between classes.

5.1.7 Measure of Aggregation – MOA

MOA is a measure of the number of attributes that are of other class types. Our experiments show that it is better to minimize the value of MOA. In other words, when we decrease the total number of attributes that are of other class types, we decrease the dependency among classes. We can think of MOA as a subset of DCC where only the number of attributes is being considered. Hence, as we work on decreasing the value of MOA, we get to decrease the value of DCC.

5.1.8 Measure of Functional Abstraction – MFA

Both our experiments and the literature emphasize the importance of increasing the value of MFA. This means that the inheritance relationship between classes is more effective when the sub-classes are using all the methods in the parent classes. In our analysis, we check which sub-classes are using a lower percentage of their parents' methods. These classes negatively affect the inheritance design property and hence we recommend that they should be either extracted from the inheritance relationship and be treated as stand alone classes or they should implement more of their parents' methods.

5.1.9 Number of Polymorphic Methods – NOP

This is the count of the methods that can exhibit polymorphic behavior. From our experiments, the less the virtual methods in a class the worse was the measure of NOP. Moreover, other metrics (namely, CIS and NOM) were negatively affected when NOP went down to the zero level in some cases. Hence, in our analysis report we recommend the reduction of virtual methods if applicable, not their complete elimination. We warn the user that a large number of virtual methods however, has a negative effect on the understandability quality attribute.

5.1.10 Class Interface Size – CIS

Our experiments confirm what is stated in the literature; that it is better to increase the value of the CIS. A high value for this indicator implies that the public methods found in the examined class diagram dominate the number of private methods.

5.1.11 Number of Methods – NOM

This metric is left to the user's judgment as it is directly proportional to the total number of classes. Therefore, if the user finds that NOM is too high with respect to the total number of classes, then the class diagram needs to be re-examined. However, our results show that the lower is the NOM the better is the class diagram as this lowers the complexity design property of the class diagram. It was difficult to set an acceptable number of methods per class. Therefore, this metric was left to the user's judgment.

5.2 Quality Attributes

In this research work, quality attributes are the most important indicators. The increase or decrease in their values leads us to discover the weak design metrics. Also, their values are important in calculating the weighted average. Therefore, they act as a double head sword with one head pointing to the design metrics values and the other to the weighted average value.

Bansiya et al [2] set formulas to calculate the values for the quality attributes. We based our quality attributes on Bansiya's Definitions (Table 2.2, Section 2.1.5.1) and Computation formulas (Table 3.2, Section 3.1).

In the following points we explain how we interpreted and managed each quality attribute in this research:

1) Reusability: signifies reusing the components found in one class diagram to another without spending much effort. In Bansiya et al's formula, Reusability has 4 main measurements: coupling, cohesion, messaging, and design size. To reach a high value of reusability, we have to have lower values of cohesion and coupling, and higher values of messaging, and design size. If we have a class diagram that consists of this combination of values, then it is easy to reuse it in another similar situation.
2) Flexibility: the ability of a design to be adapted to provide functionally related capabilities. Its formula consists of encapsulation, coupling, composition, and polymorphism, all of which are significant characteristics when we make changes in the class diagram. Hence, if our user knows that his/her class diagram is in its early phases and might need more development later, then he/she has to increase the value of flexibility by increasing the value of encapsulation, polymorphism and composition while decreasing the value of coupling.
3) Understandability: signifies how easy the class diagram is to work with. Seven design properties out of a total of eleven are involved in Bansiya et al's formula for Understandability. To enhance understandability, we try to minimize the complexity, cohesion and coupling and increase abstraction, polymorphism, design size and encapsulation. The computation of understandability results in a negative value as it measures how hard it gets to learn and understand the class diagram. This could explain the fact that in Bansiya et al's [2] quality attribute computation equation for understandability, most metrics are given a negative sign. Bansiya et al [2] expect understandability to decrease from one release to the next as a result of adding more functionality. The objective of our research however was to improve quality attributes by improving their underlying design

4) Functionality: responsibilities assigned to the classes and made available through their public interfaces. It indicates how the classes in a design could be fully utilized. Hence, we seek to increase/decrease the appropriate values of the design properties found in this quality attribute's equation (namely: cohesion, polymorphism, messaging, hierarchies, and design size) to achieve higher functionality. In our experimental work, when we distributed a dense hierarchy into smaller hierarchies of classes, the functionality improved significantly.
5) Extendibility: shows the capability of the existing classes in a design to receive new additional requirements. For this attribute, we measure the abstraction, coupling, inheritance, and polymorphism properties of the class diagram. If we have higher values of inheritance, and polymorphism and lower values of coupling and abstraction, then this class diagram is ready to accept additional improvements. In our work, extendibility improved significantly when we reduced class coupling but got worse when we decreased inheritance and polymorphism.
6) Effectiveness: this attribute signifies the design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques. We note that the weights given by Bansiya et al [2] to this quality attribute's design metrics are much smaller than those given to the same metrics in other quality attributes. Consequently, we realized from our experimental work that an increase in the values of abstraction, encapsulation, inheritance, polymorphism and composition has a milder effect on the effectiveness quality attribute than in other attributes.

The coefficients that Bansiya et. al [2] gave to each design metric in each quality attribute equation were not explicitly stated. However, the signs that they used, positive and negative, were helpful in making the above interpretations. Through our experiments, we were able to show which design metric had a stronger negative or positive effect on each corresponding quality attribute. We summarize our findings as follows:

- Reusability is positively affected whenever CIS increases or when CAM decreases.

- Flexibility is positively affected whenever DAM increases or when MOA decreases.
- Understandability is positively affected whenever DAM increases or when ANA decreases.
- Functionality is positively affected whenever NOH increases, or when CIS increases, or when CAM decreases.
- Extendibility is positively affected whenever ANA or MFA increases.
- Effectiveness is positively affected whenever ANA or MFA increases.

Table 5.1 shows the positive effects on quality attributes which result from the increase/ decrease of the design metrics. The arrows show the increase/decrease in the design metric that cause a net increase in the Quality attribute. For example, in the first column, Reusability increases with the increase in DSC and CIS, and decreases with each of the DCC and CAM.

	Reusability	Flexibility	Understandability	Functionality	Extendibility	Effectiveness
DSC	↑		↑	↑		
NOH				↑		
ANA			↑		↓	↑
DAM		↑	↑			↑
DCC	↓	↓	↓		↓	
CAM	↓		↓	↓		
MOA		↑				↑
MFA					↑	↑
NOP		↑	↑	↑	↑	↑
CIS	↑			↑		
NOM			↓			

Table 5.1: The Increase/Decrease in Design Metrics which Positively Affects the Corresponding Quality Attributes

5.3 Weighted Average

The aim of the weighted average is to help the user (optionally) express and implement the degree of importance of each quality attribute by setting his/her list of priorities. The default setting is that all six quality attributes are given equal priorities. However, the user may select the first item in the priority list according to the following:

1) If the user is interested in making the components in the examined class diagram highly reusable, then he/she should assign reusability first priority.
2) If the user does not have a complete or a clear requirement document and accordingly expects more development and changes in the class diagram, then he/she sets flexibility to be of first priority.
3) If the user is interested in an easy self-explanatory class diagram, then he/she chooses understandability as having first priority.
4) If the user needs to make sure that all classes are fully utilized in the class diagram, then he/she assigns functionality first priority.
5) If the user needs to measure the capability of the existing classes to receive new additional requirements, then extendibility may be given first priority.
6) If the user needs to ensure the design's capability in achieving the desired functionality and behavior, then he/she would give effectiveness first priority.

The remaining attributes may also be given second, third or lower priorities according to the user's interest, in a sequential manner. For example, if the user has an incomplete requirement document, then 2 important priorities (namely flexibility and extendibility) should be chosen to be of highest priority. The user has to decide on which of them gets first and which gets second priority. The rest of the quality attributes would then take equal weights (at third priority level).

The user has to make sure that when changes are made to the class diagram, the same set of priorities should be applied on all versions of the design. This makes it easier on the user to visually compare the different results of the weighted average for all runs.

The weighted average value acts as an indicator that points to the importance of quality attributes. Its effectiveness is in that it shows how the quality attribute in question is evaluated relative to the other quality attributes.

Therefore, in setting the priority for the weighted average, it is very important that the user decides on which quality attributes are most significant in the product being tested. Each quality attribute is based on a set of design metrics. If we want a certain quality attribute to be in the lead, then we have to examine the design metrics that affect this attribute. Through the SDAnalysis tool, we are able to trace these metrics and work with the user on increasing their values. Our experiments have demonstrated that we could work our way, based on calculated values, to guide the user in improving the design metrics for the individual classes.

In our experiments we applied 3 different cases of prioritization:

1) The first (Case I) was a non-prioritized list where all quality attributes were given equal weights. It acted as a frame of reference for the other two cases.
2) In the second case (Case II) we assigned each quality attribute a priority i.e. six levels of priority were used. We varied each list with every example.
3) In the third case (Case III) we gave only two quality attributes the first and second priorities, and the remaining quality attributes were given equal weights at the third priority level.

The first case where all quality attributes have equal weights provides the base case to refer to when comparing the second and third cases. In the second and third cases, the weighted average value is based on varying priorities. We tested the 3 cases with different settings of priorities on different quality attributes with each example that we studied. From our experiments we were able to prove the following:

If the first priority is given to a specific quality attribute (say A), then if the net weighted average value is higher than the value of quality attribute A, this implies that the priority selection was appropriate. Otherwise if the net weighted average value is lower than the value of the quality attribute A, then the values of the other quality attributes dominated A's value. In the latter case, the user is expected to work on improving quality attribute A.

For example, when we gave the understandability - which is usually a negative value - the first priority, the result of the weighted average became negative. This shows that the value of the understandability quality attribute is high enough to dominate the rest of quality attributes. But when Effectiveness was given first priority in another example and was dominated by other quality attributes, we worked on improving the Effectiveness design metrics (ANA, DAM, MOA, MFA and NOP) which gave a much better value for net weighted average.

5.4 Observations

The observations in the analysis report were a major part of our solution approach. It is through these observations that we are able to give the user feedback on the examined class diagram and to point out the weaknesses. The observations in the report were based on individual class analysis instead of the value of the design metrics or the value of the quality attributes. This is to save the user's time in finding the exact location of the weakness.

Only 3 design metrics were not covered by our observations. They are DSC, CAM, and NOM. As mentioned above, it was very difficult to set a threshold for the total number of classes (DSC) or the total number of methods (NOM). The number of classes in a class diagram could range anywhere from 6 classes to more than 300 classes. Therefore, this metric was left without a specific threshold, but it was used as a basis in computing other design metrics. Similarly, it was difficult to set upper and lower limits for the number of methods per class diagram. Hence, we just took the average of the total number of methods per class (total methods/num of classes) to be the value of the NOM.

As mentioned earlier, since we used a different design metric to measure the cohesion property (CAM) and this design metric was not normalized, then it was difficult to set a range for it. Therefore, we did not include a threshold for this design metric. However, it was affirmative to note that when CAM decreased it had a positive effect on the reusability, understandability, and functionality quality attributes.

The user needs to go through all observations in the report first before resolving any of them. Some classes could appear in many different observations, but when one of them is resolved, it can automatically resolve the others. Moreover, the user is required to evaluate the report against the requirements document in order to differentiate between the observations that should be dealt with and the observations that could be neglected. Therefore, although the analysis report presented to the user helps to identify the weaknesses in the classes, yet the human judgment and experience is still very much needed.

To wrap up, we conclude that the user should examine all the four elements of the analysis report (the metrics, quality attributes, weighted average and observations) before deciding whether to pass the class design to the next phase in the software development life cycle or not.

Chapter 6: Summary and Conclusion

In this chapter we summarize our research work and contributions and we point out directions for further research work.

6.1 Research Overview

The main objective of this research was to present a metrics-based solution for evaluating class diagrams which would help project managers and software quality personnel (as well as developers) quantitatively assess the class diagram. The main problem that we address in this work is to pinpoint the weaknesses in a class diagram, based on solid metrics, and give well-analyzed directions on how the user can deal with them.

Our solution is based on measuring quality attributes which are computed from class design metrics. The approach was to collect the metrics, compute the quality attribute values, analyze the metrics and finally present a report to the user. We offer the user the choice of setting priorities on the quality attributes. We applied computation formulas for the metric and quality attributes prescribed by earlier researchers but our experimental work gave us much insight into their meanings and dependencies. This enabled us to give a more concrete report to the developer which would guide him/her in improving the diagram's quality.

6.2 The Research Approach

In this thesis, we first presented our literature survey results about the following essential topics:

1. The list of metrics proposed by earlier researchers and used to measure the quality of Object-Oriented designs in general and of class diagrams in particular. We summarized the different quality metrics collected by each research and the main techniques for collecting these metrics. Also, we showed how some researches addressed the issue of visually representing their metrics.
2. The features offered by automated metric tools and how they extract, record, manage, and represent the design metrics extracted from a class diagram.

3. The quality attribute evaluation models based on different aspects chosen by earlier researchers. This evaluation could be based on experience and understanding, or on findings, or on comparison between results.
4. Prioritization techniques applied in earlier work on software quality attributes.

Based on the above findings, we presented our solution approach which went as follows:

1. We selected and computed an appropriate list of design metrics from the class design diagram which would help us assess the diagram and compute the basic product quality attributes.
2. We selected and computed the six basic product quality attributes and added a prioritization scheme for the user to set on them.
3. We suggested thresholds for most design metrics. We were able to compare the extracted metrics from the examined class diagram against these thresholds and subsequently assess the class diagram and each individual class.
4. Through a visual tool that we developed, we offered the user the list of quality attributes and a means to arrange them in a priority setting that would match his/her desired product quality.
5. We calculate a weighted average value for the prioritized quality attributes based on formulas set by earlier researchers.
6. We present a list of observations based on the computed values for each design metric and the overall quality attribute values.
7. We assist the user in identifying the weaknesses in the class diagram and how he/she can go about resolving them.

We used a CASE tool (IBM Rational Software Development Platform version 6.0) for drawing the class diagram and a ready-made tool for extracting some of the class diagram metrics (SDMetrics). The output file from the SDMetrics was the input to our visual tool (SDAnalysis) in which we compute the remaining class diagram metrics and interact with the user to set priorities and generate the analysis reports.

6.3 Research Contribution

Our research contributions may be summarized in the following outcomes:

1. The extension of the work of earlier researchers on class design metrics and quality attribute modeling.
2. The addition of a prioritization scheme for product quality attributes.
3. The addition of a design metrics analysis layer and generation of analysis reports.
4. The refinement of the analysis report based on experimental results.
5. An interactive visual tool for the computation of some metrics, setting of priorities and generation of analysis reports.

6.4 Directions for Further Work

The quality of object-oriented designs has become one of the major concerns of both researchers and industry personnel. They both seek a high quality class diagram structure which would lead to high quality code. This in turn would save time and effort during the later phases of the software development life cycle.

We believe that the work done in this thesis paves the way for further research, specifically in the following directions:

- More in-depth analysis of the intricate relations between the software product quality attributes.
- Optimization of the range of values for each quality attribute's metrics with respect to the other attributes' metrics which may sometimes have contradicting priorities. This would lead to the addition of another set of observations that can help the user arrive at an optimized overall quality attribute level.
- Creation of a repository of reports for each project which would keep record of the first class diagram and its modifications. This would give another dimension to the analysis reporting layer in which a comparison of the effects of changes applied may be stored and tracked.
- Building a database of normative values for the thresholds which the user may be allowed to set. Further research work may suggest hypotheses about the best

- The establishment of a framework of quality metrics and quality attributes evaluation based on our solution approach.
- Enhancement of the SDAnalysis tool to track the repository of class diagrams and analysis reports from earlier versions of the same class diagram.

Appendix A: the SDAnalysis Tool

As the aim of our research is mainly to help the quality assurance manager in evaluating a class diagram prior to implementation, we had to develop a means to communicate with our user. We developed the SDAnalysis tool to read the extracted values of the class metrics (available from the SDMetrics tool) for a class diagram drawn in Rational Rose and interact with the user in setting quality attribute priorities. The SDAnalysis tool computes the additional metrics and quality attribute values and generates an analysis report about the overall quality of the class diagram. The end result that we present to our user is a weighted average for all the quality attributes listed in table 3.2 and an analysis report about the diagram. Figure A.1 shows the scenario between the designer and the quality assurance manager:

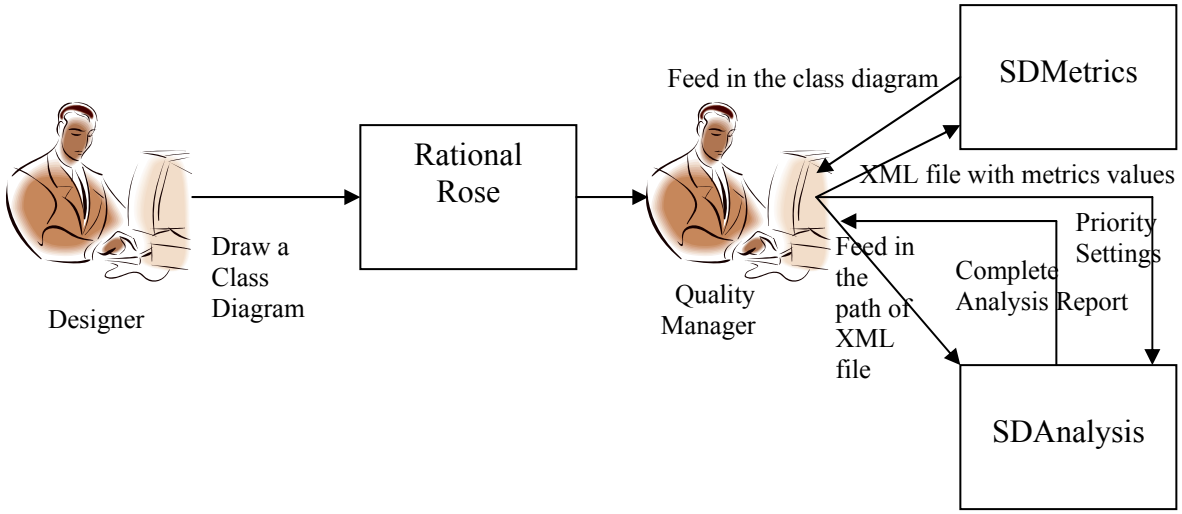


Figure A.1: The scenario between the designer and the quality assurance manager

Figure A.2 shows the scenario when the quality assurance manager is not satisfied with the output analysis report:

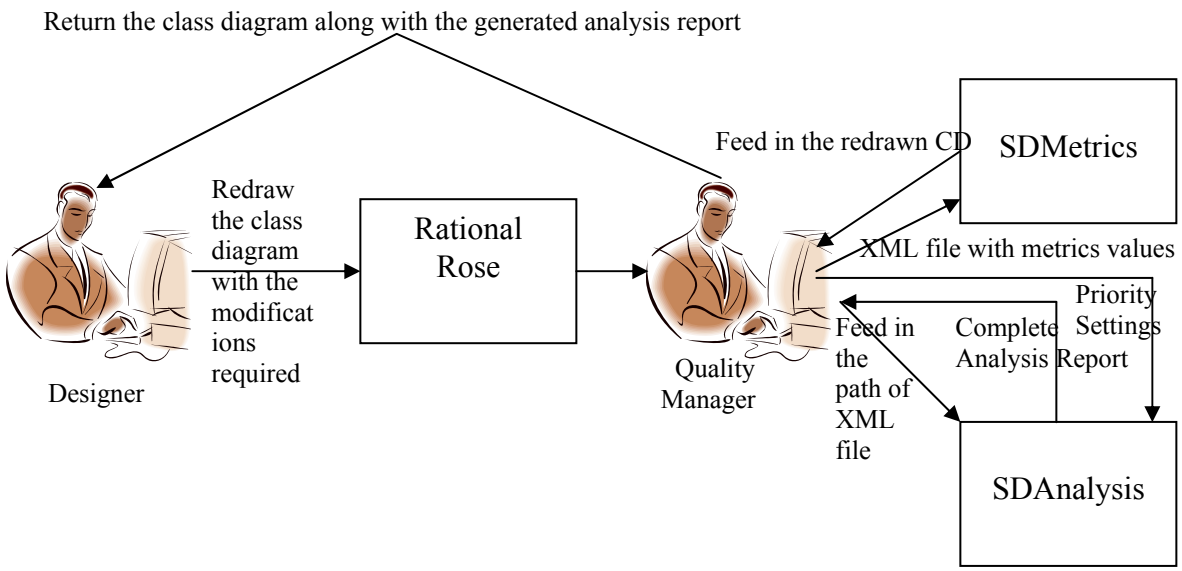


Figure A.2: The scenario for modifying the class diagram according to the analysis report

Figure A.3 is a use case diagram summarizing the services offered by the SDAnalysis tool.

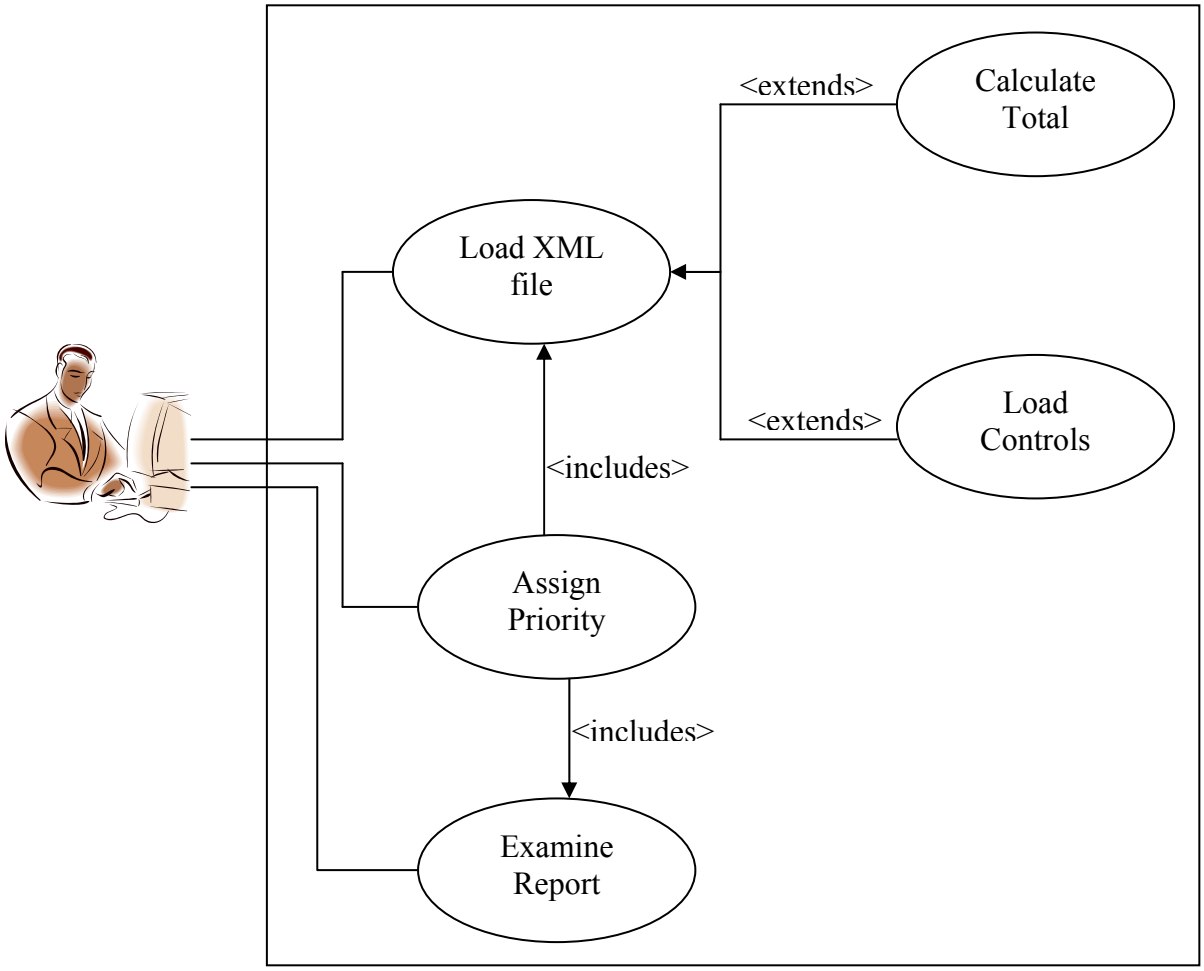


Figure A.3: Use Case Diagram for the SDAnalysis Tool

In Figure A.4 we show the class diagram of the interactive tool which shows an overview of our classes and the relationships among them. Followed by a detailed explanation for each class separately.

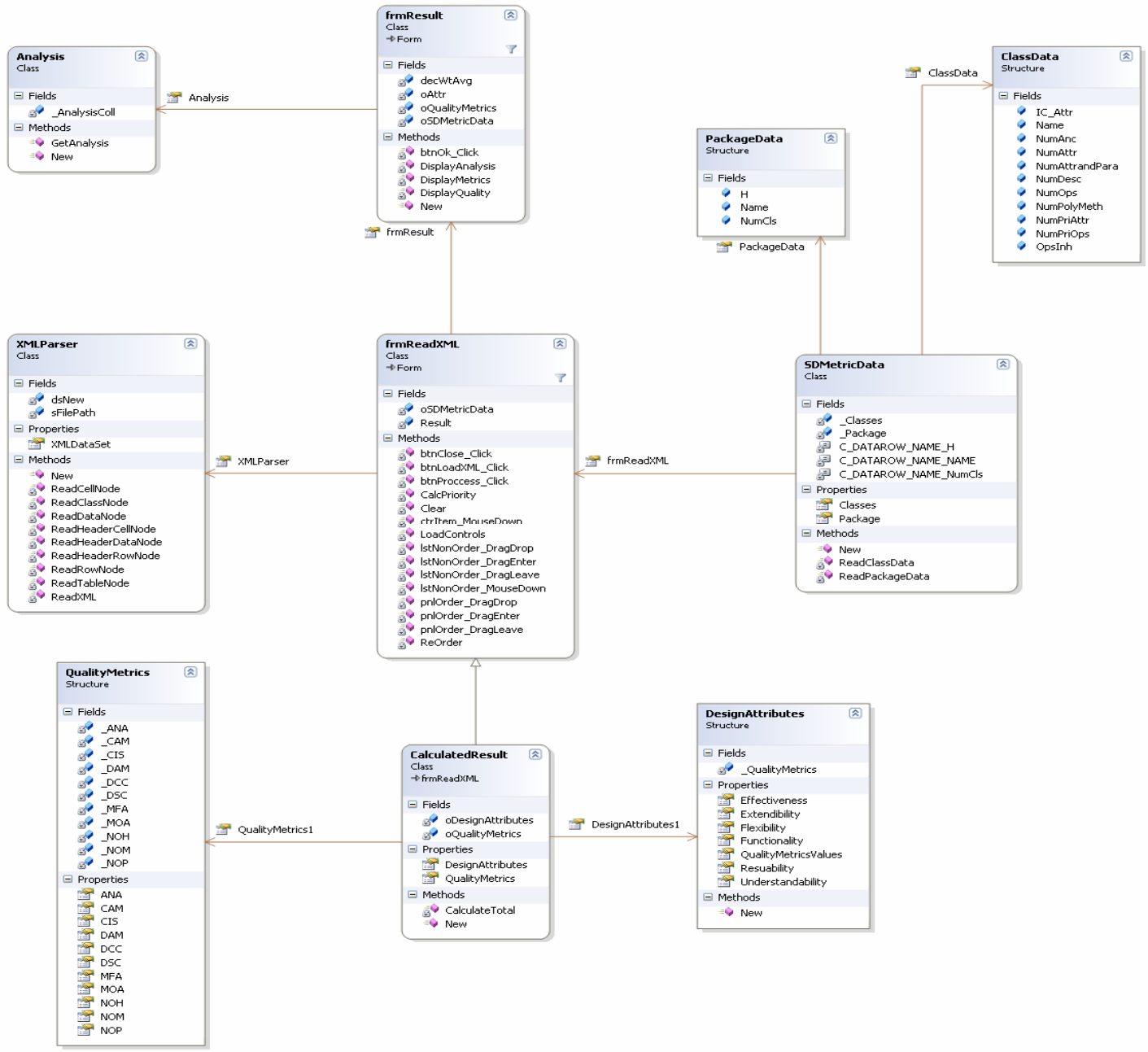
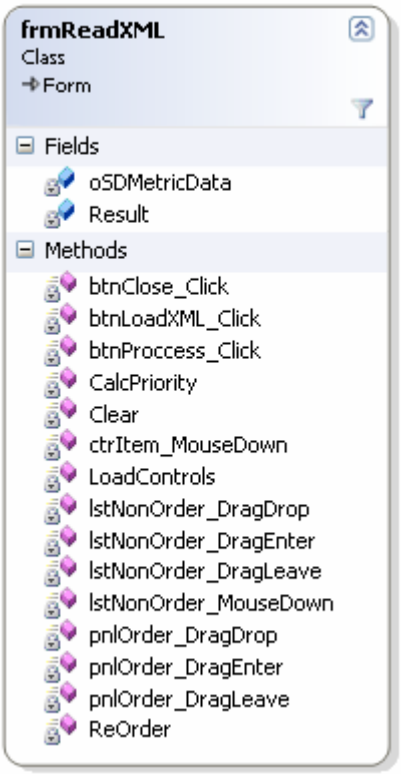
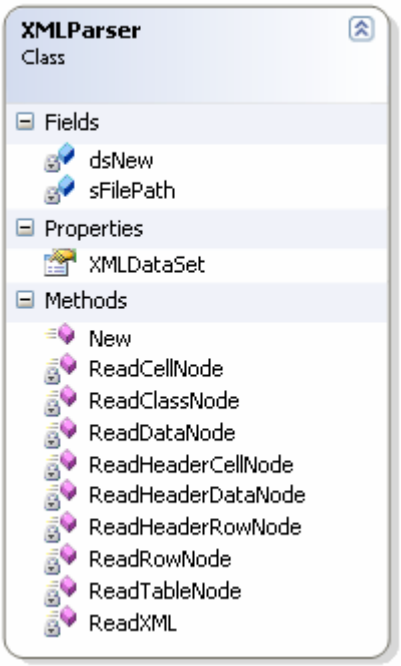


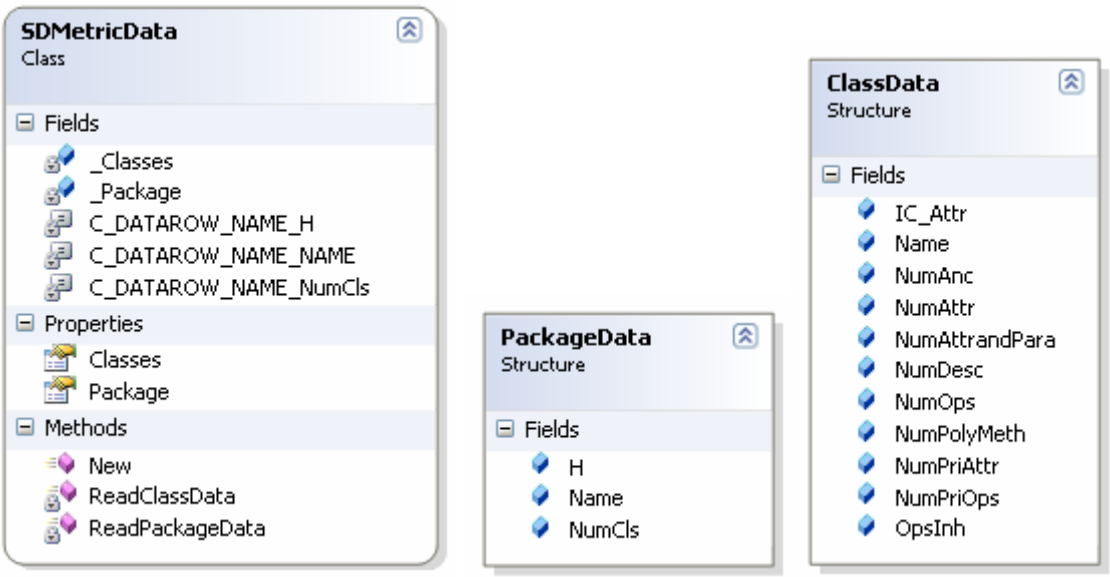
Figure A.4: The class diagram for SDAnalysis Tool



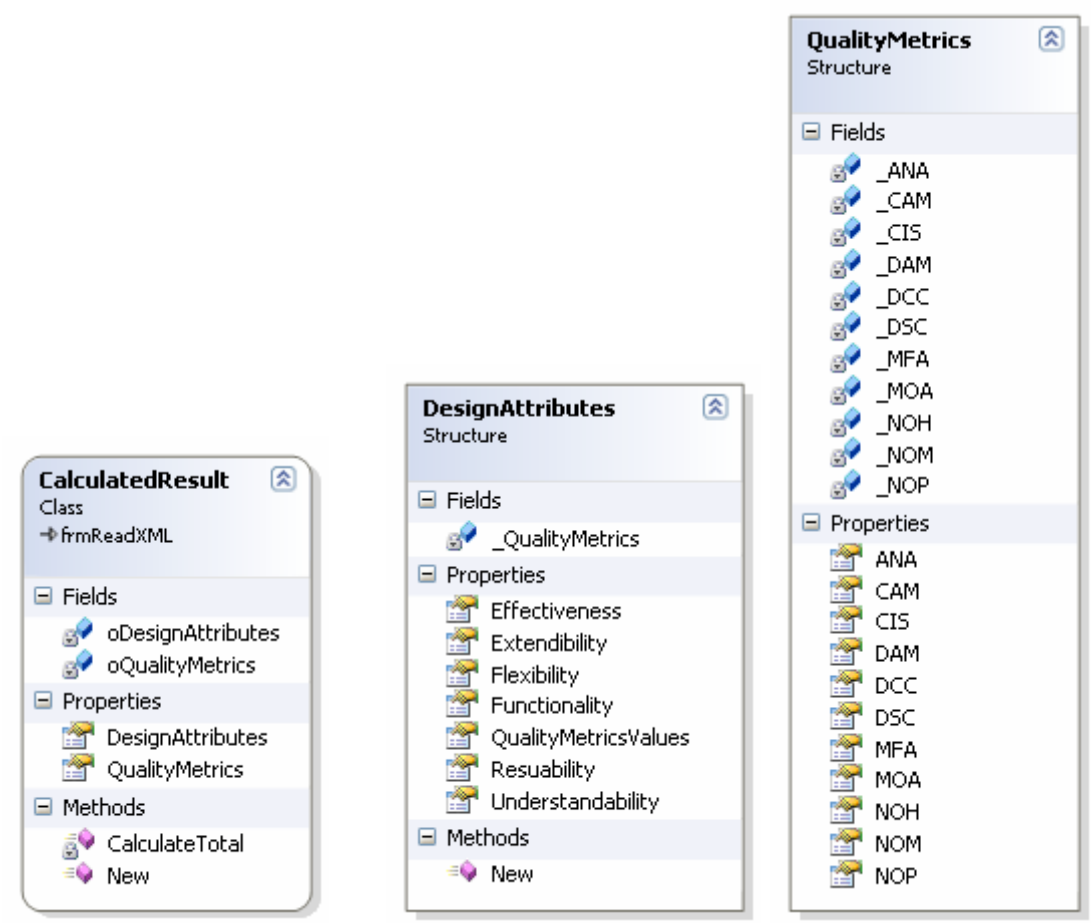
The frmReadXML class is the control class in the tool. It takes in the location and name of an xml file, sends the file to the XMLParser class and takes back from it a data set that consists of the items found in the xml file. It then sends this data set to the SDMetricData class to divide the items in the data set into PackageData and ClassData. The frmReadXML class sends the divided data to the CalculatedResult class to calculate the design metrics and quality attributes. Also, this class (frmReadXML) loads the priority for the quality attributes and calculates their weights. Finally it sends the weighted average and the results of the design metrics and quality attributes to the frmResult class.



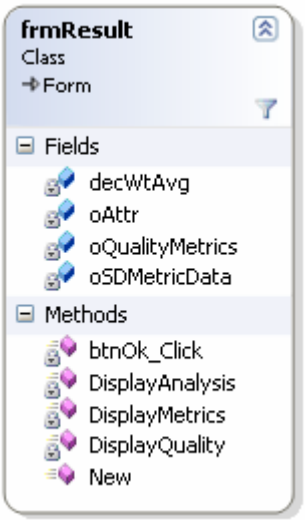
This class (XMLParser) takes in the path of an xml file, then reads the data in the file and return a data set with all the data it read. It reads the xml file in a sequential manner and saves its reading in a table form.



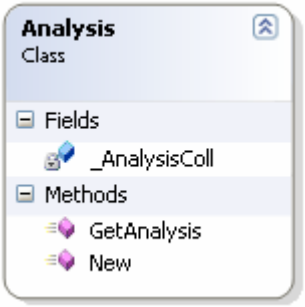
SDMetricData is a class responsible to break down the information found in the data set into PackageData and ClassData. The information found in the fields of these two structures is what gets used in carrying out the calculations.



The frmReadXML class calls CalculatedResult class and sends to it an instance of SDMetricData. The CalculatedResult class calculates the quality metrics and design attributes and saves their values in the appropriate structure.



The frmReadXML class finally sends the calculated weighted average according to the priority list, the calculated design metrics and quality attributes, and the raw values of the package and classes. The frmResult class displays the values of the design metrics, quality attributes, and the weighted average value. Also, it displays the observations for each metric based on the evaluation of the thresholds conducted within the Analysis class.



The Analysis class checks the status of each metric for each class and accordingly returns the appropriate message to the frmResult class.

Appendix B: Permission for Use of SDMetrics

From: Juergen Wuest (SDMetrics) <info@sdmetrics.com>
To: "Dalia Kamal A. Rizk" <drizk@aucegypt.edu>
Date: Wed, Mar 12, 2008 at 2:38 PM
Subject: Re: Your SDMetrics Academic License Request
mailed-by sdmetrics.com

Hello,

thanks for your interest in an SDMetrics academic license.

Please find below the conditions of the SDMetrics academic license. If you agree with these conditions, please reply to this e-mail stating that you accept the SDMetrics Academic License. You will then receive the SDMetrics full version by e-mail (Zip archive, 490 KBytes).

Best regards,

Juergen Wuest
SDMetrics Academic license

The conditions of the regular license apply to you (see http://www.sdmetrics.com/FullLic.html for conditons). In particular, it follows from the regular license that - you may use SDMetrics on commercial or non-commercial projects of your own or your industry partners' (i.e., measurement of commercial and non-commercial systems developed by you or your industry partners), and you may charge the industry partner for your services, SDMetrics is to be used by you, or staff/students of your department under your supervision, on computer systems of your department of your organization, you MAY NOT install or use SDMetrics at an industry partner's site, or have staff members of the industry partner use your copy of SDMetrics. If this is a necessity, the industry partner is required to purchase a regular license.

Your Additional Obligations

Publications and presentations of empirical studies using SDMetrics must contain an acknowledgment which mentions the name "SDMetrics" and the URL "http://www.sdmetrics.com", for example: "Design measurement was performed with SDMetrics, available at http://www.sdmetrics.com."

My Additional Rights

For promotional purposes, I may quote published quantitative results from your studies using SDMetrics on the SDMetrics website, brochures, and flyers.

I may include the name of your department/organization on a list of customers of SDMetrics. For promotional purposes, I may post this list on the SDMetrics website, brochures, and flyers.

And a request

If possible, please place a link to the URL "http://www.sdmetrics.com" in a suitable location at your department's, project's, or personal web site. I'd appreciate if you could thus help promote SDMetrics.

Appendix C: MFC Library Version 7.0

For the sake of testing the relationship between the total number of classes and the depth of inheritance, we tested on a close depth of inheritance (six levels) to our third example, but within a much larger structure of a class diagram. We selected a part of the Microsoft Foundation Class Library Version 7.0 [9]. Figure C.1 shows the part of the class diagram for MFC Library Version 7.0.

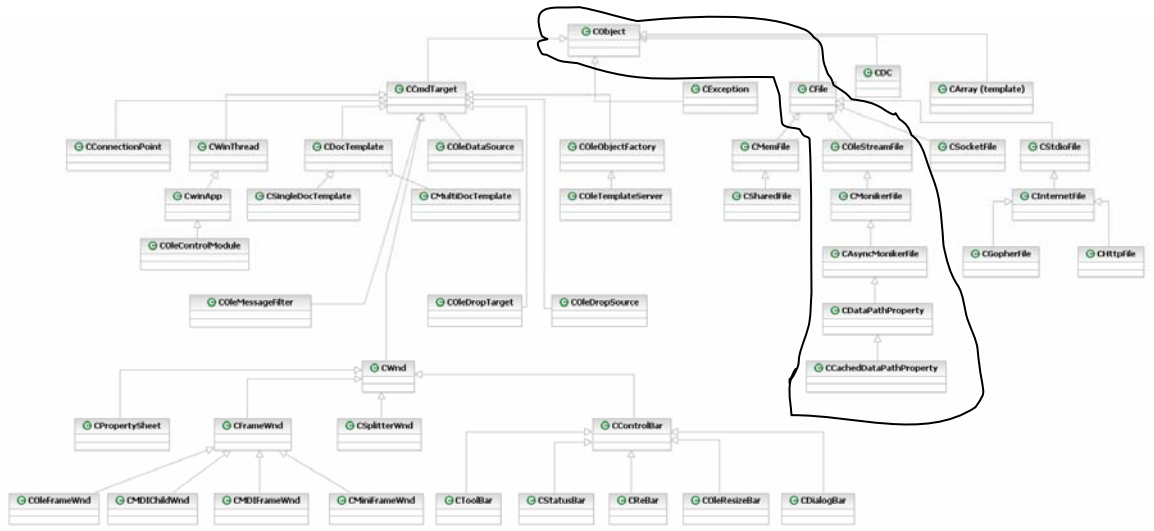


Figure C.1: Part of MFC Library V 7.0

The part in the freeform curve marks a six-level hierarchy, but the drawn part of MFC library class diagram consists of 45 classes. Therefore the value for NOH is 1. Also, the value for ANA is 2.82 which is far from half the worst value for ANA which is in this case should be 11 from (0.5*(990/45)).

Therefore, according to the total number of classes in the diagram, if the depth of inheritance exceeds half the worst value for ANA (which is △DSC/DSC), then this will affect understandability, extendibility, and effectiveness quality attributes.

References

1. Anselmo, D. and Ledgard, H. "Measuring Productivity in the Software Industry," *Communications of the ACM*, vol. 46, no. 11, pp. 121-125, 2003.
2. Bansiya, J. and Davis, C. G. "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4-17, 2002.
3. Basili, V.R., Briand, L.C., and Melo, W.L. "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, 1996.
4. Bhatti, S.N. "Why Quality? ISO 9126 Software Quality Metrics (Functionality) Support by UML Suite," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, 2005.
5. Chidamber, S.R., and Kemerer, C.F. "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20., no. 6, pp. 476-493, 1994.
6. Fenton, N. E. and Pfleeger, S. L. Software Metrics: A Rigorous and Practical Approach Second Edition. PWS Publishing Company, 1997.
7. Frakes, W. and Terry, C. "Software Reuse: Metrics and Models," *ACM Computing Surveys*, vol. 28, no. 2, pp.415-435, 1996.
8. Harrison, R., Counsell, S. J., and Nithi, R. V. "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491-496, 1998.
9. Hierarchy Chart (MFC). Available at: <http://msdn.microsoft.com/en-us/library/ws8s10w4(VS.71).aspx>. (last accessed: April, 2009).
10. IEEE Std. 1061-1998, "IEEE Standard for a Software Quality Metrics Methodology," revision, *IEEE Standards Dept.*, Piscataway, N.J., 1998.
11. Karlsson, J. "Software Requirements Prioritizing," *Proceeding of the 2nd International Conference on Requirements Engineering (ECRE'96)*, IEEE, pp. 110-116, 1996.
12. Khan, R.A. and Mustafa, K. "Metric Based Testability Model for Object Oriented Design (MTMOOD)," *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 2, 2009.

13. Kitchenham, B., Pfleeger, S. L., and Fenton, N. "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, vol. 21, no. 12, pp. 929-944, 1995.
14. Kostecki, J. A. "Book Review: Object-Oriented Software Metrics by Mark Lorenz and Jeff Kidd" *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 1, pp.91-93, 1995.
15. Lakshminarayana, A., Newman, T. S., Li, W., and Talburt, J. "Automatic Extraction and Visualization of Object-Oriented Software Design Metrics," *Proceedings of SPIE – The International Society for Optical Engineering*, vol. 3960, pp. 218-225, 2000.
16. Lamouchi, O., Cherif, A.R., and Levy, N. "A Framework Based Measurements for Evaluating an IS Quality," *Proceeding of the 5th Asia-Pacific Conference on Conceptual Modelling (APCCM 2008)*, pp. 39-47, 2008.
17. Lee, K., and Boehm, B. "Value-Based Quality Processes and Results," *Proceedings of the third workshop on Software quality*, ACM, pp. 1-6, 2005.
18. Liu, K., Zhou, S., and Yang, H. "Quality Metrics of Object Oriented Design for Software Development and Re-development," *Proceedings of the First Asia-Pacific Conference on Quality Software (APAQS'00)*, IEEE, pp. 127-135, 2000.
19. Ogasawara, H., Yamada, A., and Kojo, M. "Experiences of Software Quality Management Using Metrics through the Life-Cycle," *Proceedings of the 18th International Conference on Software Engineering (ICSE '96)*, IEEE, pp. 179-188, 1996.
20. Pressman, R. Software Engineering: Practitioner's Approach Fifth Edition. McGraw-Hill International, 2001.
21. Schneidewind, N. F. "Body of Knowledge for Software Quality Measurement," *IEEE Computer*, vol.35, no. 2, pp. 77-83, 2002.
22. Sharma, A., Kumar, R., and Grover, P.S. "Estimation of Quality for Software Components – an Empirical Approach," *ACM SIGSOFT Software Engineering Notes*, vol. 33, no. 6, 2008.
23. SimDrug–model description. Chapter 10. SimDrug: A Multi-Agent System Tackling the Complexity of Illicit Drug Markets in Australia. Available at: <http://epress.anu.edu.au/cs/html/ch10s03.html>. (last accessed: April, 2009).

24. Siragi OCR project, uml design diagrams. 18 May 2006. Available at: <http://siragi.sourceforge.net/design/class-diagram.png>. (last accessed: April, 2009).
25. Sommerville, I. Software Engineering Sixth Edition. Addison-Wesley, Pearson Education Limited, 2001.
26. Stein, C., Etzkorn, L., and Utley, D. "Computing Software Metrics from Design Documents," *ACM Southeast Regional Conference*, pp. 146-151, 2004.
27. Stiglic, B., Heričko, M., and Rozman, I. "How to Evaluate Object-Oriented Software Development?" *ACM SIGPLAN Notices*, vol. 30, no. 5, pp. 3-10, 1995.
28. Succi, G., and Liu, E. "A Relations-Based Approach for Simplifying Metrics Extraction," *ACM SIGAPP Applied Computing Review*, vol. 7, no. 3, pp.27-32, 1999.
29. Swokowski, E., Olinick M. and Pence, D. Calculus Sixth Edition. Boston: PWS Publishing Company, 1994.
30. Tanaka, T., Aizawa, M., Ogasawara, H., and Yamada, A. "Software Quality Analysis & Measurement Service Activity in the Company," *The 20th International Conference on Software Engineering*, IEEE, pp. 426-429, 1998.
31. UML. Sun Developer Network (SDN). Available at: <http://gceclub.sun.com.cn/prodtech/javatools/jsenterprise/learning/tutorials/jse8/uml_class_diagram.html>. (last accessed: April, 2009).
32. Wagner, S. and Deissenboeck, F. "An Integrated Approach to Quality Modelling," *Fifth International Workshop on Software Quality (WoSQ' 07)*, IEEE, 2007.
33. Wagner, S., Deissenboeck, F., and Winter, S. "Managing Quality Requirements Using Activity-Based Quality Models," *International Conference on Software Engineering*, ACM, pp. 29-34, 2008.
34. Wang, H. "Software Metrics Collection Techniques for Product Assessment," Available at: http://www.comp.nus.edu.sg/~wanghao/new/ic52a5_frameVersion.htm, 1999. (Accessed: March, 2003).
35. Weisstein, Eric W. Triangular Number. From MathWorld--A Wolfram Web Resource. Available at: <http://mathworld.wolfram.com/TriangularNumber.html>. (last accessed: April, 2009).
36. Wüst, J. SDMetrics. Available at: <http://www.sdmetrics.com/index.html>. (last accessed: April, 2009).