CHAPTER 3

SOFTWARE ASSESSMENT MODEL FOR OBJECT ORIENTED STRUCTURE

3.1 INTRODUCTION

System engineers analyse and understand the business of the proposed system by analyzing the user requirement documents and by studying the detailed design document of the system. The use of models as an effective means of understanding the interrelationships among different aspects and requirements of the system is as old as engineering. In this chapter, a Software Assessment model and a Software Assessment tool is proposed to evaluate the software by applying design metrics.

3.2 THE EXISTING MODELS

Everything has to be measured to evaluate and eventually improve the quality of the design. The gap between the quality that is perceived and assessed through measurement has to be bridged and improved even at the design level. The chapter starts with a presentation of the Software Assessment Model approach and also includes a discussion on the limitations of the existing quality models.

Design plays a key role in the development of software. The quality of design is crucial and is a fundamental decision element in assessing the software product. The early availability of Design quality evaluation provides a better way to decide the quality of the final product. This avoids presumption in the quality evaluation process. Hence software metrics provide a valuable and objective insight for enhancing each of the design
quality characteristics. The research proposes a quality model to assess the Software Assessment of any object-oriented system based on the works of (Chidamber & Kemerer 1994); (Chidamber & Kemerer 1991); (Chidamber et al. 1998) and (Basili et al. 1996) suggests two new metrics. The research focuses on analyzing a set of metrics, which has a direct influence on the quality of the software and creating a metrics tool based on java that can be used to validate the object-oriented projects against these metrics. The analysis is carried out on a set of real world projects designed using Unified Modeling Language, which are used as test cases. These metrics and models are proposed to add more quality information in refining any object-oriented system during the early stages of design itself.

3.2.1 Factor-Criteria-Metric Quality Models (FCM)

The speed of production is meaningless without an assessment of product quality. This observation has led software engineers to develop models of quality whose measurements can be combined with those of productivity.

One of the most frequently used quality models is the decomposition approach that is used both (McCall et al. 1977) and (Boehm et al. 1978) commonly known as the Factor-Criteria-Metric (FCM) quality model. FCM models are usually constructed in a tree-like fashion. The upper branches hold important high-level quality factors related to software products, such as reliability and maintainability, with which the developers would like to quantify. Each quality factor is composed of lower-level criteria, such as structuredness and conciseness. These criteria are easier to understand and they measure the factors themselves by the actual metrics proposed for them. The tree describes the relationships between factors and criteria and hence one can measure the factors in terms of the dependent
criteria measures. This notion of divide and conquer has been implemented as a standard approach for measuring software quality (ISO-9126 1991).

**Limitations of FCM Quality Models**

Although this approach is cited throughout the whole software engineering literature and is implemented in several commercial CASE tools, the two main drawbacks that limit its usability are the obscure mapping of quality criteria into metrics and the poor capacity to map quality problems to causes. The drawbacks of the FCM quality models are in essence due to the fact that quality is mapped to a large set of measurements that are only implicitly related to the rules and principles of good design. These drawbacks can be eliminated if the quality of code and design would be explicitly assessed in terms of deviations from a set of quantifiable good design principles and heuristics.

**3.2.2 Hybrid Approach to Quality Models**

In (Sahraoui & Boukadoum 2001) and (Sahraoui et al. 2001) the authors comment on the FCM models and proposed an improvement to such predictive quality models. The authors emphasize that the use of precise threshold values and their interpretation in the absence of formal models and the crudeness of the derived rules which can only serve to build naive models are the two diseases of the current approaches for building predictive models. The authors propose a novel approach by building fuzzy decision processes that combine both software metrics and heuristic knowledge (Basili et al. 1996) from the field. The authors claim that this hybrid approach would improve the efficiency of quality prediction and provide a more comprehensive explanation of the relationship that exists between the observed data and the predicted software quality characteristics.
A critical view on the model reveals that while the fuzzification of threshold values seems applicable and well found (Fenton & Neil 2000), the second part containing decomposition of heuristic knowledge is far from being traceable. In addition to that, no case studies are provided in the model so that the practicability of the approach is not yet proved. As a conclusion, it is firmly believed that this approach is useful in the future especially on the side of a proper parameterization of the interpretation models of the metrics. Yet it does not offer a comprehensive approach for an improved quality model.

3.2.3 Factor-Strategy Quality Models (FS)

The approach is intended to improve the FCM (Marinescu 2005) paradigm with respect to the two major drawbacks discussed earlier. The Factor Strategy Quality Model (FS) is used for the assessment of software quality that decomposes quality into a set of high level quality factors and maps each factor to a set of strategies that quantify deviations from rules of good design. The detection strategy in FS quality model is captured as a quantifiable expression of a rule by which design fragments that are conforming to rule can be identified in the source code.

In the FS quality model, the future work indicated the issues of threshold values. The model provides some answers on the question of how to define and set the threshold values for the individual measurement and only some improvements are visible.

3.2.4 Quality Model for Object Oriented Design (QMOOD)

The QMOOD (Bansiya & Davis 2002) (Quality Model for Object Oriented Design) is a comprehensive quality model that establishes a clearly defined and empirically validated model. The model assesses the OOD
quality attributes, such as understandability and reusability and relates it through mathematical formulae, with structural OOD properties such as encapsulation and coupling. Also, it presents the OOD metrics needed to quantify these OOD properties.

The QMOOD model consists of six equations that establish the relationship between six OOD quality attributes as reusability, flexibility, understandability, functionality, extendibility and effectiveness and eleven design properties shown in Figure 3.1. The measurement for these design properties is also a part of QMOOD.

For example, reusability is a function of the coupling measure, the cohesion measure, the messaging measure and the Effort Estimate. The coupling measure is quantified using Direct Class Coupling (DCC) metric. The cohesion measure is quantified using Cohesion among Methods in a Class (CAM) metric. The Messaging metric is quantified by Class Interface Size (CIS) metric. Finally, the Effort Estimate is quantified by the Effort Estimate in Classes (DSC) metric. All these are measurable directly from class diagrams and applicable to UML class diagrams.

Object Oriented Design Model (OODM)

The OODM (Object Oriented Design Model) (Reißing 2001) is another approach proposed by Reibing. This approach is based on introducing a new formal model that captures the UML class diagram elements. This model is an abstraction layer defined on the top of the UML meta-model. In OODM, the layers make metrics definitions easier to read and comprehend. However, this approach lacks the precision required, since the proposed layer consists of plain English expressions.
Evaluating the existing approaches for finding an appropriate syntax for representing metrics, leads to the elicitation of some desirable properties that should be met by any potential language. This language should be a standard language with a formal syntax to gain wide acceptance among practitioners. It should be easy to express the metrics in terms of such a language. Also, it should be straightforward to automate metrics collection defined in this language.

The DM Crawler data model proposed (EI Wakil et al. 2005) expresses metrics as XQuery expressions that targets XMI documents. The approach views the design documents as data and metrics as queries of the data. This approach lacks to investigate the relevant metrics formalization need for dealing with static structure of the model and its behaviour.

3.3 STRATEGY FOR PREDICTING SOFTWARE QUALITY

Based on the survey of the existing object-oriented Software Quality models, a set of properties that should be exhibited by any OOD
quality model is proposed, which is of practical use. Lacking any of these properties will result in an inapplicable quality model. The model objectives and the quality characteristics to be assessed should be stated explicitly. Some models (Kim & Boldyreff 2002); (Marchesi 1998); (Lorenz & Kidd 1994); (Lanza 2003) just introduce metrics without stating how these metrics could be used to assess quality. For the proposed model, new metrics and some well-known metrics are adopted and validated.

We regard the analysis / design / implementation phase as a series process in which the information about software product gradually increases as the process progresses. Some of the metrics can be applied to the design specification at the initial design phase, and some of them can be applied at the later design phase. Based on the fact, as design phase progress, we approximate the design improvements only using the applicable metrics to the design specification.

3.3.1 Need For The Model

The ideas on what is considered as good or poor design by different developers will likely fluctuate. However, being aware of these differences should allow the organization to define good software design. Simultaneously it should help the less skilled programmers to produce software with better design. We address this issue by our design phase model with the tool-based approach. The proposed Software Assessment Model identifies and assesses the software by applying design metrics improve the quality of the design.

This new approach ensures to develop the quality of Software Assessment Model. More precisely, this approach is based on detection strategies that quantify rules directly related to the quality of design. This new
quality model may be regarded in a way as the end-result of all the mechanisms and techniques defined in the accomplishment of the initially stated goal of this dissertation. The proposed Software Assessment Model is an attempt to quantify the object-oriented metrics with a real time procedure for better verification and validation of the metrics. The High Level Software Quality Indicator applies the concept to UML designed object-oriented projects, which are then translated to their respective skeletal source code using a standard design suite like Rational. The translated source code using UML is given as the input to the Software Assessment Tool which then analyses and applies the Software Assessment quality model to assess their conformance to the object orientedness. The model intimates the design imperfection at an earlier stage and generates a lot of productivity to the project management team through its real time validation. The model also calculates the redesigning efforts by thus providing a valuable reference for the deviations and the efforts.

Initially after identifying the design metrics, the object-oriented design is converted into UML diagram by using the Rational Rose case tool. The case tool generates the output in an object-oriented programming language. The output of the Rational Rose case tool is taken as the input to the proposed tool. The Software Quality Indicator is developed using these results, which points out the various design imperfections. Essentially, the model attempts to bring in improvements needed to achieve the best possible results, in terms of efficiency, effectiveness and flexibility.

To perform the Software Assessment validation accurately, design document details have to be collected based on identical requirements, which may be developed using an object-oriented programming language. The projects are implemented using the sequential life cycle model, a well known object-oriented analysis design method with Rational Rose Case Tool and
JAVA programming language. The result proves to be a significant predictor of fault proneness and the proposed tool would represent a good early quality indicator for the object-oriented design metrics in the design stage. The Software Assessment tool and Quality Indicator can be applied to industries that develop small scale applications compared to the various large scale applications based on the object-oriented methodology.

3.3.2 Object Oriented Construction

Today most of the software industries focus on developing software through object-oriented programming. The three main features that are important in any object-oriented software development are the data encapsulation, reusability and polymorphism. Object-oriented programming is stressed, because of its usefulness in the object-oriented concepts. The development of an application, ranging from simple to complex, depends on the programmer’s style and practice. Any programmer adopts his own method for writing algorithms, procedures or coding and modifies it until the developer feels that the design is perfect. The model can assess any system design developed using the object-oriented concepts and UML.

3.3.3 Analyzing Object Oriented Quality

A design can be stated valid, if it satisfies the object-oriented criteria such as coupling, cohesion, reuse, hierarchy structure, polymorphism and the property inheritance scheme (Lorenz & Kidd 1994); (Li-Thiao-Té et al. 1998); (Tegarden et al. 1995). However, few researchers identify the wrong usage of object-oriented design concepts where a metric has been provided to permit its detection (Li-Thiao-Té et al. 1998); (Barnes & Swim 1993). Usually, a suite of coherent metrics is recommended to be used within a measurement programme, however, the dependencies between them are still unclear (Henderson-Sellers 1996).
This work is aimed at understanding the object-oriented concept mechanisms and the effects from the viewpoint of designers modelling strategy by applying suitable metrics. The design metrics can assess the uncertainty at the early stages of the design and after a precise identification of suspected problems with valid metrics for its assessment and suggestions are made for design improvement.

3.3.4 Proposed Model

The proposed Software Assessment Model as shown in Figure 3.2 identifies and assesses the software design by applying design metrics at high-level design.

Figure 3.2 Software Assessment Model
The Software Assessment Model helps to correct the imperfections identified during the Software Assessment and to improve the quality of the design. The approach to Software Assessment Model is that, after identifying design metrics, the object-oriented design is converted into UML diagram by using Case Tool, which generates the code. The output from the Case tool is given as an input to the Software Assessment Tool. Software Assessment Tool evaluates the design structure and by using the results, the Software Quality Indicator is developed.

3.3.5 Adaptation of Object Oriented construction to UML Representation

In the proposed model, the system that has been developed using the object-oriented structure has to be converted to UML. The Unified Modeling Language (UML) was adopted by the Object Management Group (OMG) ending the so-called ‘OO methods war’. Since then it has become the de facto specification standard graphical language for specifying, constructing, visualizing and documenting object-oriented software systems, business modelling and other non-software systems (Object Management Group 1999). UML has been intensively used by software developers since its introduction. Many organizations are using UML as a common language for their project artifacts and have adopted UML as their organization’s standard. As the amount of UML models produced within an organization increased, a need for measuring their characteristics has also arisen.

3.3.6 Class Diagram Selection

A great effort has been made in the field of software measurement in order to achieve better quality software. But most of the metrics pursue the goal of evaluating by means of quantitative measures which evaluate the quality of the final product. In order to ensure quality, the focus should be on
measuring the quality characteristics of early artifacts, such as class diagrams which are based on those measurements thereby obtaining a prediction model in the early life cycle for object-oriented programming quality characteristics (ISO/IEC 9126-1.2).

As class diagrams constitute a key artifact in the early development of object-oriented programming, the effort expected on improving their quality is likely to pay off many times over in the later phases. It is in this context, where software measurement plays a vital role, because the early availability of metrics could contribute in evaluating the class diagram quality in an objective way avoiding bias in the quality evaluation process.

In response to the great demand of the measures for measuring the quality characteristics of class diagrams, such as maintainability and after a thorough review of some of the existing object-oriented measures that can be applied at a high level design stage (Chidamber & Kemerer 1994); (Abreu & Carapuça 1994) a set of measures is proposed for assessing the design quality. The focus on UML is considered as a standard in object-oriented modeling (Object Management Group 1999). Since the external quality characteristic can be evaluated for a finished product or nearly finished product, the model focuses on measuring the internal quality characteristics and the structural complexity of class diagrams and sequence diagram are represented in Figure 3.3 and Figure 3.4. The idea is to use those measures to predict class diagram maintainability early in the object-oriented programming development life-cycle.
Figure 3.3 An Example of Class Diagram

Figure 3.4 An Example of Sequence Diagram
3.3.7 Translation of Design to Code

In the development of object-oriented software, the class diagram is a key early artifact that lays the foundation of all later design and implementation work. The early focus on class diagram quality may help software designers build better object-oriented software, without unnecessary revisions at later development stages when changes are more expensive and more difficult to perform. It is the place, where software measurement plays an important role. The early availability of metrics contributes quality evaluation to class diagram in an objective way avoiding bias in the quality evaluation process. Moreover, metrics provide a valuable and an objective insight into specific ways of enhancing each of the software quality characteristics. The object-oriented structure design as discussed is represented in UML using Rational Rose is depicted in the following Figure 3.5.

![Figure 3.5 Sample Class Diagram in UML](image-url)
The conversion of the class diagram into different code level can be performed with the different options in the Rational software tool which is shown in Figure 3.6.

![Figure 3.6 Tool Selections in UML](image)

The rational design model supports many tools like Ada, ANSI C++, CORBA, J2EE, JAVA, ORACLE and VB. The proposed model works
on the Java/J2EE platform, in which the high level design in the form of class diagrams can be converted to source code as shown in Figure 3.7.

Figure 3.7 Code Selections from Design
3.3.8 Assigning Design as Input

The Software Assessment Model accepts the input from design because of the fact that an object-oriented programming system has become more popular in the field of software development. There is also a strong interest in automating the production of these metrics. Recently CCCC (C and C++ Code Counter), JMetric (www.it.swin.edu.au/projects/jmetric) and (McCabe & Butler 1989) metrics tool are popular among them. Most of these new metrics and tools, however, deal only with the language-dependent source code. This is typically available at the later stages of the software life cycle and fails to address the importance of the software artifacts produced during the earlier stages such as requirement and analysis stages. This model reports an attempt to overcome these limitations by measuring UML models in the design stage is shown in Figure 3.8.

![Figure 3.8 Code from Design Stage](image)
3.4 THE PROPOSED TOOL

In order to reduce the time spent on problem detection, tools are needed to automate the detection to a large extent. An overview of the Software Assessment Tool and their operation sequence is depicted in Figure 3.10 and Figure 3.11. From these tools perspective, the following steps are implied in the detection process. First, the source files are parsed and the design information is extracted. This step detaches the approach from the lower abstraction level of the concrete implementation and brings it to the higher level of abstraction offered by a design model.

This model is used as an abstraction layer towards the various programming languages that support object-orientation. Second, the metrics used for the detection are computed. These metrics are implemented for SQL queries which are based on the tables containing the design model. Having the tables and the implementation of the metrics, the measurement results can be computed.

The results are manually investigated based on the detection strategy. Software Assessment Tool extracts design information from the design level and stores it in the form of tables. These tables contain information about all the classes, methods and variables defined in the analysed project and about the inheritance relations among classes. It delivers information about the method calls and the variable accesses. Compared to other similar tools, it extracts substantially more information and is more stable.

3.4.1 Architecture of the Tool

The Architecture of the Software Assessment Tool shown in Figure 3.10 illustrates the following modules namely a File acceptor module,
Parser controller, Project controller and the Metrics Display module. The File acceptor module is controlled by the UI controller and the command controller to fetch the source properly. The Parser controller processes the input source file and generates ASCII character stream, which is then converted into tokens. The tokens are then filtered for method, variable and class counts. The result is aggregated by the Project controller module in the Block through sorted vector, String Collector and Array Data. The statistics generated by the module is presented by the result module through various means like a chart or a drill table or saving as a file or printing the results.

### 3.4.2 Development of the Software Assessment Tool

The Software Assessment Tool Figure 3.9 reads source data from the File Stream and validates the input data against policy, which is defined to be a proper java file or project (more java files). It also establishes a stream level parser to read the token based on metrics and scores the token in a data collector class variable for further analysis. It provides flexible Jtable feature to list the class and method data for verification. If the mode chosen is Project aggregate, the results of each analysis in data collectors provide either a default chart comparison for class and method data or saves the information in a file with a user defined name for future reference or prints the information using a standard printer. It also provides the facility to display various metrics.
Figure 3.9 Software Assessment Tool

The Software Assessment model also accepts the project profile from the user like the team size like small, medium, big and also the team experience in technology like average, good, excellent and calculates the redesigning efforts cost for project manager’s reference about the productivity of the design team.
The primary goal of software system quality engineering is to supply a prime quality wares through the employment of some specific techniques and processes. One of the strategies is applying data processing techniques to software system metric and defect information collected throughout the software system development process to spot potential low-quality program modules. This application will predict the software system defects by victimization of ranking technique. Here, the tester and developers will predict the software system defect and assigns the priority of the defective modules according to the defect severity. And it will predict the price estimations. Value estimation within the sense we are able to predict the point in time for the full software system development method for the actual

**Figure 3.10 Software Assessment Tool Architecture Diagram**

The primary goal of software system quality engineering is to supply a prime quality wares through the employment of some specific techniques and processes. One of the strategies is applying data processing techniques to software system metric and defect information collected throughout the software system development process to spot potential low-quality program modules. This application will predict the software system defects by victimization of ranking technique. Here, the tester and developers will predict the software system defect and assigns the priority of the defective modules according to the defect severity. And it will predict the price estimations. Value estimation within the sense we are able to predict the point in time for the full software system development method for the actual
software system. And it will predict the testers and developers wage details and time.

Many object-oriented style metrics are developed to assist in predict software system defects or measure style quality. Since a defect prediction model might provide crucial clues concerning the distribution and placement of defects and, thereby, take a look at prioritization, correct prediction will save prices within the testing method.

3.4.3 Defect Prediction

The first step to create a prediction model is to come up with instances from computer code archives like version management systems, issues following systems, and so on. Every instance will represent a system, a computer code part (or package), an ASCII text file, a class, an operation (or method), and/or a code amendment per prediction graininess. Associate instance has many metrics (or features) extracted from the computer code archives and is tagged with buggy/clean or the amount of bugs. Associate initialized report generator is whose purpose is generating report knowledge into a computer program (Excel sheet). This includes developer details like name, ID, Designation comment written at commit time etc.,

3.5 METRICS FOR DESIGN IMPROVEMENT

In this section, the metrics used for design description is reviewed and correlated with quality attributes. Then, the metrics that are considered for changeability prediction in the Software Assessment Model is explained in the Figure 3.11. These metrics measure the structural complexity of UML class diagrams due to the use of relationships, such as associations, generalizations, aggregations and dependencies. The model also considers the traditional metrics such as, the number of classes and the number of attributes.
Design characterization is mostly done through metrics. A conventional distinction is made between the architectural or high-level design and algorithmic or the low level design. According to Rombach, architectural (high-level) design properties have more influence on maintainability than algorithmic (low-level) design (Rombach 1990). In the realm of object-oriented design, numerous design metrics have been published (Abreu & Carapuça 1994); (Chidamber & Kemerer 1991).

One suite of object-oriented design metrics has been proposed by Chidamber and Kemerer and progressively defined (Chidamber & Kemerer 1994); (Chidamber & Kemerer 1991); (Chidamber et al. 1998). The suite, characterizes a class and its relationships with the other classes of the system and comprises four inter-class metrics, DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling Between Objects), RFC (Response For a Class), two intra-class metrics, WMC (Weighted Methods per Class) and LCOM (Lack of Cohesion in Methods).

Several studies have been conducted to validate the metrics and to relate them to some maintainability property. Studies (Li & Henry 1993); (Li et al. 1995) showed that the C&K metrics plus some other metrics are good predictors of the maintenance effort. Basili et al. (1996) showed that the metrics are predictors of fault proneness. Some studies on the relationship between design and maintainability are based on the other design metrics. Hsia et al. (1995) for example, studied the effect of architecture on maintainability and found that maintainability is better for systems with broader trees, shallower inheritance trees. Briand et al. (1997) defines 18 coupling measures between classes and found that some of the coupling metrics are significant predictors of fault-proneness.
To summarize, several design metrics, in particular the C&K metrics suite can be considered as good predictors of some aspects of maintenance, yet changeability is not one of the aspects considered. In this work, the Software Assessment Model has utilized from C&K metrics as candidates for changeability indicators. C&K metrics are solidly defined, theoretically well grounded, widely used and have already been shown to be maintainability predictors. Since the LCOM metrics does not seem to be relevant to maintainability (Basili et al. 1996), it was also given importance in the model. In calculating WMC, the model assumed the method complexity to be the same for all methods, which is assumed by most authors who previously used WMC.
Figure 3.11 The collection of Quality Attributes to Metrics for Design Improvement
3.5.1 Traditional Metrics

There are many metrics that are applied to traditional functional development. From experience in object-oriented software development, three of these metrics are applicable to object-oriented development, such as complexity, size and readability.

**METRIC 1: Size**

Size of a class is used to evaluate the myth of understanding of the code by developers and maintainers. Size can be measured in a variety of ways. These include counting all physical lines of code, the number of statements, the number of blank lines and the number of comment lines. Lines of Code (LOC) counts all lines. Non-Comment and Non-Blank (NCNB) is sometimes referred to as the source lines of code and counts all lines that are not comments and not blanks. Executable Statements (EXEC) is a count of executable statements regardless of the number of physical lines of code.

Executable statements are the measure which is least influenced by the programmer or the language style. Thresholds for evaluating the meaning of size measures vary depending on the coding language and the complexity of the method. However, since size affects ease of understanding by the developers and maintainers, classes and methods of large size will always pose a higher risk.

3.5.2 Object Oriented Specific Metrics

Many different metrics have been proposed for object-oriented systems. The object-oriented metrics that are chosen measure principle structures. If the object-oriented metrics chosen is improperly designed, it negatively affects the design and code quality attributes.
Tool support is considered as a critical factor in this research because of the huge amount of data used in JDK release. The metrics that are published in literature are automatically collected from design stage and can be used. The selected metrics, which are used as independent variables in DPT, are as follows:

The Chidamber and Kemerer (C&K) (EI Emam et al. 2001); (Li & Henry 1993) metrics suite is a widely used metrics suite. It has been validated by several researchers and it forms the core of this study. However, researchers identified some deficiencies in some of the C&K metrics. Churcher & Shepperd (1995) pointed out potential problems with some of the C&K metrics and suggest alternatives. Hitz & Montazeri (1996) argued that it is premature to apply C&K metrics because of the unresolved uncertainty about the precise definitions of many quantities to be observed and their impact upon subsequent indirect metrics. Li’s (Li 1998) suite of object-oriented metrics solved deficiencies in some of the C&K metrics and has alternatives to some of their measures. Finally, the C&K metrics was selected because of its wide acceptance among the software engineering community and numerous previous empirical studies have also used them.

The selected object-oriented metrics are primarily applied to the concepts of classes, coupling and inheritance. A class is a template from which objects can be created. This set of objects shares a common structure and a common behaviour manifested by the set of methods. A method is an operation upon an object and is defined in the class declaration. A message is a request that an object makes of another object to perform an operation. The operation executed as a result of receiving a message is called a method. Cohesion is the degree to which methods within a class are related to one another and work together to provide well-bounded behaviour. Effective object-oriented designs maximize cohesion since it promotes encapsulation.
Coupling is a measure of the strength of association established by a connection from one entity to another. Classes are coupled when a message is passed between objects or when methods declared in one class use methods or attributes of another class. Inheritance is the hierarchical relationship among classes that enables programmers to reuse previously defined objects including variables and operators.

**METRIC 2: Weighted Methods per Class (WMC)**

The WMC is a count of the methods implemented within a class or the sum of the complexities of the methods the method complexity is measured by cyclomatic complexity. The measurement of calculating the method complexities is difficult to implement, since not all methods are accessible within the class hierarchy due to inheritance. The number of methods and the complexity of the methods involved is a predictor that requires much time and effort to develop and maintain the class. The larger the number of methods in a class, the greater the potential impact on children. Children inherit all of the methods defined in the parent class. Classes with large numbers of methods are likely to be more application specific and thus limiting the possibility of reuse.

**METRIC 3: Response for a Class (RFC)**

The RFC is the count of the set of all methods that can be invoked in response to a message to an object of a class or by some method in the class. This includes all the methods accessible within the class hierarchy. This metric looks at the combination of the complexity of a class through the number of methods and the amount of communication with other classes. The larger the number of methods that can be invoked from a class through messages, the greater is the complexity of the class. If a large number of
methods are invoked in response to a message, the testing and debugging of the class becomes complicated since it requires a greater level of understanding on the part of the tester. A worst case value for possible responses will assist in the appropriate allocation of testing time.

**METRIC 4: Lack of Cohesion in Methods (LCOM)**

Lack of Cohesion in Methods (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes. A highly cohesive module is stand-alone. High cohesion indicates good class subdivision. Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process. High cohesion implies simplicity and high reusability. Classes with low cohesion could probably be subdivided into two or more subclasses with increased cohesion.

**METRIC 5: Coupling Between Object Classes (CBO)**

Coupling Between Object Classes (CBO) is a count of the number of other classes to which a class is coupled. It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends. Excessive coupling is detrimental to modular design and prevents reuse. The more independent a class is the easier to be reused in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult. Strong coupling complicates a system since a class is harder to understand, change or correct by itself if it is interrelated with other classes. Complexity can be reduced by designing systems with the weakest possible coupling between classes. This improves modularity and promotes encapsulation.
METRIC 6: Depth of Inheritance Tree (DIT)

The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes. The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit making it more complex to predict its behaviour. Since more methods and classes are involved, deeper trees constitute greater design complexity, but result in greater potential for reuse of inherited methods. A support metric for DIT is the Number of Methods Inherited (NMI). Here, the real time practical induced inheritance is considered. The user defined complexity in inheritance is measured and an attempt is made to standardize the complexity involved in the depth of inheritance.

METRIC 7: Number of Children (NOC)

The number of children is the number of immediate subclasses subordinate to a class in the hierarchy. It is an indicator on the potential influence a class can have on the design and on the system. The greater the number of children, the greater the likelihood of improper abstraction of the parent and may be a case of misuse of sub-classing. When the number of children is greater, the reuse is greater, since inheritance is a form of reuse. If a class has a large number of children, it may require more testing of the methods of that class, thus increasing the testing time.

Using several metrics together can help managers and senior designers, who are unable to review design materials for the entire application, to exercise some measure of architectural control over the evolution of an object-oriented application. The users could by means of the WMC, DIT and NOC metrics check whether the application is getting ‘top
heavy’ too many classes at the root level declaring many methods or using the RFC and CBO metrics check whether there are interconnections between various parts of the application that are unwarranted. The metrics values are likely to change as a project proceeds from design to the implementation phase.

If the system has been well architected, the class hierarchy will be stable and the WMC, NOC, DIT metrics will reflect this characteristic. However, during implementation, new class coupling and communication may develop, affecting the CBO and RFC metric values. If implementation requires changes in the class definitions itself, the WMC and LCOM metrics will also change. Tracking these metrics is throughout the life of the project, will provide the managers with information to monitor object-oriented systems evolution. As maintenance of the architectural integrity of an application becomes an important managerial responsibility, the Software Assessment model is used as a tool to meet this challenge.

To conclude, nine metrics (WMC, DIT, NOC, CBO, RFC, LOC and WAC) along with two new metrics (Effort Estimate and Cost Estimate metrics) are considered. The latter two are discussed in the next chapter.

The Software Assessment Tool represented in Java is easy for analysis and the proposed Software Assessment model can be applied to real time designs. The application of this tool is simple and its easiness to use significantly improves the developer’s ability to identify, analyse, fix and improve quality characteristics of an object-oriented software design and implementation. The tool’s design repository allows for significant assessment of a design with other designs from related and comparable domains.
3.6 SUMMARY

This chapter discussed the proposed Software Assessment Model with its internal working. The process of design evaluation by the model was discussed with examples. The internal architecture of the tool along with its four modules viz, file acceptor, parser controller, project controller and metrics display module were explained in detail. The implementation and the coordination of the four modules in generating the metrics value which was used for reverse engineering was also discussed in detail.