CHAPTER 1

INTRODUCTION

This chapter introduces the existing software engineering particles and the methodologies used for software development. It gives an overview of the motivation, the objective and the scope of the research work carried out. It also discusses the importance of design in object-oriented system development.

1.1 SOFTWARE ENGINEERING

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. The generic model of the software engineering layers is illustrated in the Figure 1.1.

Figure 1.1 Software Engineering Layers
1.2 SOFTWARE QUALITY

Software Quality Assurance (SQA) is an umbrella activity that is applied throughout the software process (Owens & Khazanchi 2009). SQA encompasses:

(1) A quality management approach
(2) Effective software engineering technology
(3) Formal technical reviews
(4) A multi-tiered testing strategy
(5) Document change control
(6) Software development standard and its control procedure
(7) Measurement and reporting mechanism

Quality refers to measurable characteristics of software. In general there are two types of quality control

Quality design: The characteristics that designers specify for an item. It includes the requirements, specifications, and the design of the system.

Quality of conformance: The degree to which the design specification are followed. It focuses on implementation based on the design.

Quality Control

Quality control is the series of inspections, reviews, and test used throughout the develop cycle of a software product. Quality control includes a feedback loop to the process. The major objective of the quality control is to
minimize the produced defects, increase the product quality. They are several Implementation approaches to control the quality of the software:

1. Fully automated
2. Entirely manual
3. Combination of automated tools and human interactions

Quality assurance consists of auditing and reporting function of management. The major is to provide management with the necessary data about product quality. The gain is to insight and confidence of product quality.

**Cost of Quality**

A cost of quality includes all costs incurred in the pursuit of quality for perform quality related work. Quality cost includes:

**Prevention cost**

- Quality planning
- Formal technical reviews
- Testing equipment
- Training

**Appraisal cost**

- In-process and inter-process inspection
- Equipment calibration and maintenance
- Testing
Failure cost

Internal failure cost

- Rework, repair, and failure mode analysis

External failure cost

- Complaint resolution
- Product return and replacement
- Help line support
- Warranty work

Three import points for quality measurement:

- Use requirements as the foundation
- Use specified standards as the criteria
- Considering implicit requirements

Software Engineering process

- The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.
- Process defines a framework that must be established for effective delivery of software engineering technology.
- The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products are produced, milestones
are established, quality is ensured, and change is properly managed.

**Methods**

- Software engineering methods provide the technical aspects for building software.
- Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.
- Software engineering methods rely on the set of modeling activities and other descriptive techniques.

**Process & Generic Process Model**

A software process is illustrated in Figure 1.2 and it is defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created.

Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.
Figure 1.2 Generic Process Model

Each framework activity is populated by a set of software engineering actions.

Each software engineering action is defined by a task set that identifies the work tasks that are to be completed, the work products that will be produced, the quality assurance points that will be required, and the milestones that will be used to indicate progress. A generic process framework for software engineering defines five framework activities such as communication, planning, modeling, construction, and deployment.

Process Flow

This flow process in the Figure 1.3 describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.
Figure 1.3 Process flows in SE process

A linear process flow is illustrated in Figure 1.3 (a) which executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.

An iterative process flow is illustrated in Figure 1.3 (b) which repeats one or more of the activities before proceeding to the next.

An evolutionary process flow is illustrated in Figure 1.3(c) which executes the activities in a “circular” manner. Each circuit through the five activities leads to a more complete version of the software.

A parallel process flow is illustrated in Figure 1.3(c) which executes one or more activities in parallel with other activities.
Traditional models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams, since the quality of the software plays a major important role.

So the development of the software system is measured based on the quality of the system. To make the software a standardized product, it is important to express the characteristics of the system in an objective and qualitative terms. Some of the standard methodologies are used by many developers and companies to measure the quality of the software system.

1.3 SOFTWARE ENGINEERING METHODOLOGIES

Methodologies, in general are categorized as the sequential and cyclical, which are also informally known as the waterfall and spiral methodologies respectively. The methodologies are generic in design and have been simplified to emphasize a key aspect. In a sequential methodology, the four phases like analysis, design, implementation and testing follow one another sequentially. In a cyclical methodology, the four phases such as analysis, design, implementation and testing are cyclic, with each cycle generating an incremental contribution to the final system. The computer software industry has a major confusion in terms of naming these methodologies. The Boehm-Waterfall methodology is most often quoted as the sequential methodology. Likewise, the Boehm-Spiral methodology is most often quoted as the cyclical methodology, which possesses the characteristics like the sequential methodology, with many stages.
Sequential Methodology

The sequential methodology, which is widely known as the Waterfall model, consists of the following phases such as analysis, design, implementation and testing. The analysis phase comes first, followed by the Software Assessment, the implementation phase and finally the testing phase as shown in Figure 1.4. The term that denotes each phase may be different and there may be a management decision point at each phase transition.

Cyclical Methodology

A cyclical methodology, also known as the spiral model, addresses some of the problems introduced by the sequential methodology. A cyclical methodology has four phases. The methodology iterates over the processes by consuming less time at each phase as shown in Figure 1.5. The document structure and deliverable type from each phase has an incremental change in the structure and content with each cycle or iteration. More detail is generated as the methodology progresses. Finally, after several iterations, the product is complete and ready to ship. The cyclical methodology may continue shipping multiple versions of the product. Ideally, each phase is given equal attention.

Figure 1.5 Cyclical Methodology

In the Boehm-Spiral software engineering methodology, the process spirals from stage to stage, with each spiral getting closer and closer to a final solution. However, the Boehm-Spiral software engineering methodology also has a steady progress from one stage into the next stage with an explicit review between each stage. However, in engineering practice, the term spiral is used as a generic name to any cyclical software engineering methodology, which includes cycles leading to prototypes and multiple versions.

1.4 OTHER ESTABLISHED METHODOLOGIES

Object-Oriented Programming (OOP)

OOP is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. A feature of objects is that an object's procedures can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self"). In OOP, computer programs are designed by making them out of objects that interact with one another. There is significant diversity of OOP languages, but the most popular ones are class-based,
meaning that objects are instances of classes, which typically also determine their type. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that really care about are the objects to manipulate rather than the logic required to manipulate them.

Although object oriented software development methodologies have become ubiquitous in software engineering circles, a brief look at the basic definitions and the history of their evolution is necessary for understanding the motivations behind this thesis, and the basis upon which it builds. A brief overview of the Method Engineering discipline is also presented, mainly in order to clarify the position of this thesis in regard to the discipline, and also to highlight the distinctions that separate this thesis from current Method Engineering practices.

The Booch Methodology

At present, there are many established software engineering methodologies. The Booch (Grady 1994) software engineering methodology provides an object-oriented development in the analysis and Software Assessment. The analysis phase is splitted up into fewer steps. The first step establishes the requirements from the customer’s perspective and generates a high-level description of the system's function and structure. The second step, domain analysis, accomplishes its tasks by defining object classes, state diagrams, their attributes, inheritance and methods and is validated. The analysis phase is completed with a validation step. The analysis phase iterates between the customer's requirements, the domain analysis step and validation, until consistency is reached between these steps. Once the analysis phase is completed, the Booch software engineering methodology develops the architecture in the Software Assessment.
The Software Assessment is iterative. A logical design is mapped to a physical design, where the details of execution thread, process, performance, location, data type, data structure, visibility and distribution are established. A prototype is created and tested. The process iterates between the logical design, physical design, prototype and testing. The Booch software engineering methodology is sequential in a way that, the analysis phase is completed first and then the Software Assessment. The methodology is cyclical because each phase is composed of smaller cyclical steps. There is no explicit priority setting or a non-monotonic control mechanism. The Booch methodology concentrates on the analysis and Software Assessment and does not consider the implementation or testing phase in detail.

**Object Modeling Technique (OMT)**

The Object Modeling Technique (OMT) (Derr 1995); (Loomis et al. 1987) is a well known example for a software engineering methodology. The OMT methodology (Tryfona et al. 1997) deals with the object-oriented development in the analysis and Software Assessment. The analysis phase starts with the problem statement which includes a list of goals and a definitive enumeration of key concepts within a domain. This problem statement is then expanded into three views or models namely an object model, a dynamic model and a functional model. The object model represents the artifacts of the system. The dynamic model represents the interaction between these artifacts represented as events, states and transitions. The functional model represents the methods of the system from the perspective of data flow. The analysis phase generates the object model diagrams, state diagrams, event flow diagrams and the data flow diagrams. The system Software Assessment follows the analysis phase, where the overall architecture is established. The system is organized into subsystems which are
then allocated to processes and tasks, taking into account the concurrency and collaboration.

The persistent data storage is established along with a strategy to manage shared global information. The boundary situations are examined to help guide tradeoff priorities. The system Software Assessment is followed by the object Software Assessment, in which the implementation plan is established. Object classes are established along with their algorithms with special attention to the optimization of the path to the persistent data. Issues of inheritance, association, aggregation and default values are examined. In each phase, a cyclical approach is opted. The OMT is very much like the Booch methodology, where emphasis is placed on the analysis and Software Assessments for initial product delivery. Both the OMT and Booch methodology does not emphasize on implementation, testing or other life cycle stages.

**Rational Objectory Methodology**

The Rational Objectory is a full life cycle software engineering methodology. Rational Objectory is an iterative process governed by the requirements of the management. Rational Objectory activities create and maintain models to aid the developer in supporting the methodology. The Rational Objectory software engineering methodology (Seo & Choi 2000) can be described in two dimensions such as the time and process components. The time dimension represents the dynamic aspect of the process and is expressed in terms of cycles, phases, iterations and milestones. The process component dimension is described in terms of process components, activities, artifacts and workers.
Responsibility Driven Design

In later years, Wirfs-Brock developed the Object Oriented Design (OOD) modeling approach called Responsibility Driven Design (RDD) (Wirfs-Brock et al. 1990). Wirfs-Brock work is built upon the Class Responsibility Collaborators (CRC) work of Beck and Cunningham. For each class, different responsibilities (roles and actions) are defined. To fulfill the responsibilities of each class, it is necessary to demonstrate the collaboration with other classes. CRC cards provide a physical artifact for the designers to manipulate from which designers can actually see how the classes collaborate with each other. The participants are allowed to gain a firsthand experience on how the system works by physically picking up the cards and playing the role of that class. CRC cards are simple enough for novices to use, but many times CRC cards lack the extensibility necessary to model complex systems.

1.5 OBJECT ORIENTED DESIGN

The necessity to deal with the increasing complexity of software programs was the essential factor that influenced the evolution of programming paradigms. Object-oriented programming (Timothy 2008) provides a set of proper mechanisms for the management of complexity, namely data abstraction, encapsulation, modularity, inheritance and polymorphism. Rumbaugh et al. (1991) proposed an implementation method in which programs are organized in object collections that cooperate among themselves, where each object representing an instance of a class, each class is a part of a class hierarchy and all the classes are related through their inheritance relationships (Alfonseca 1990).

Data Abstraction An abstraction expresses the essential characteristics that make an object different from the other object.
Abstractions offer a precise definition of the object’s conceptual borders from an outsider’s point of view.

**Encapsulation** It is used for separating the “contractual” interface from its implementation. It is the process of splitting the elements that form the structure and behaviour of an abstraction into individual compartments.

**Modularity** The classes and objects obtained after the abstraction and encapsulation process must be grouped and then deposited in a physical form called a module. The classes and objects that result after the logical level design can be viewed as physical containers. These modules form the program’s physical architecture. A program can be split into a number of modules that can be compiled separately but that are coupled among them.

**Inheritance** defines a relation among classes in which a class shares its structure and behaviour with one or more classes.

**Object Hierarchy**

Aggregation is the relationship between two objects, in which one object is a part of another object. From a semantic point of view, aggregation indicates ‘a part of’ relationship. For example, it can be said that “a wheel is a part of a car”, since there is a relation between a wheel and a car.

**Polymorphism** The option of using an object in another object’s stead when both objects share the same interface is called polymorphism. Polymorphism is therefore one of the fundamental concepts of object-oriented programming.
### 1.5.1 Object Oriented Development Life Cycle

Object orientation has a profound effect on the software development life cycle, largely through successful exploration of the capabilities inherent in the paradigm. Two of the principal effects are as follows:

- The concept of an object forces a different conceptualization of the problem and solution domains and requires different analysis techniques.

- The property of encapsulation allows projects to proceed in an incremental fashion, where increments are far more insulated from each other than was previously possible.

It is beyond the scope of this dissertation to detail the variety of object-oriented development methodologies that have been proposed in the recent years. Fortunately, the state of the practice has evolved to the point where a high degree of commonality has emerged between techniques, making it possible to present the salient points of most methodologies in an overview fashion. The objective is to provide an overview of understanding as how the recent life cycle methodologies differ from the traditional life cycle methodologies. The following text and diagram in Figure 1.6 are adopted from the Software Architects’ Synthesis (SASY) (c) development life cycle description (McGregor & Korson 1994).

There are five main phases in the application development process like,

**Domain Analysis:** It is the process of coming to an understanding of the domain of the application. The domain is modeled, clarified,
documented and fundamental classes and their interrelationships are sorted out.

**Application Analysis:** It is the process of determining exactly what is built for the requested application. The requirement specifications are clearly spelled out in this phase.

**Application Design:** It is the process where the mechanism for implementing the actual application is determined. System architecture, data structures, efficiency considerations, etc., are all dealt with here.

**Class Development:** In this phase, classes are developed and tested in the language of implementation.

**Incremental Integration and System Testing:** Classes are integrated into clusters and subsystem modules and finally merged into a complete application. Integration testing verifies that these modules work correctly, effectively and meet requirements.

Many of these phases or steps are familiar from a traditional structured design process. The major difference is that all steps are performed repeatedly, as necessary, in multiple iterations. A number of iterations result in a complete, testable module or sub-module termed as an increment. This iterative process of developing system increments continues until the system is complete. As later increments are implemented, it is inevitable that misconceptions and outright errors will be detected in increments developed earlier. Due to the insulating effect of encapsulation, all types of incorrect system behaviour are corrected, when discovered, with much less violence to the overall design than would normally occur in a traditional development environment.
One complete increment of several iterations is diagrammed below.

- An increment is a defined subset of the total system functionality.

- Iteration is defined as one pass over some set of activities.

Application design takes a very prominent role over coding, in the object-oriented life cycle. Some of the object-oriented methodologies do not even mention coding as a separate phase. The increased prominence of the design task also means that the level of detail available at the design level in an object-oriented life cycle is much greater than what is available in a traditional system design.

The iterative nature of the object-oriented product life cycle also presents metric opportunities and techniques that are not available with traditional production methods. Since analysis and design are tasks in the life cycle that are repeated throughout the life of the product, the opportunity exists for a class of early metrics. Object-oriented designs have been determined to be conceptually more stable than other techniques because the major system concepts usually do not change from early stages of the design. Thus estimates made at an early point in development can be refined, rather than discarded. The complete system is assembled from multiple increments.
19

1.5.2 Overview of Object Oriented Structures

![Diagram of Object Oriented Development Life Cycle]

**Figure 1.6 Object Oriented Development Life Cycle**

Not all activities require equal effort for different iterations.

**Figure 1.7 Pictorial Description of Object Oriented Terms**


<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Defines the structural properties of classes, unique within a class, generally a noun.</td>
</tr>
<tr>
<td>Class</td>
<td>A set of objects that share a common structure and behaviour manifested by a set of methods. The set serves as a template from which object can be instantiated (created).</td>
</tr>
<tr>
<td>Cohesion</td>
<td>The degree to which the methods within a class are related to one another.</td>
</tr>
<tr>
<td>Coupling</td>
<td>Object X is coupled to object Y if and only if X sends a message to Y.</td>
</tr>
<tr>
<td>Inheritance</td>
<td>A relationship among classes, wherein an object in a class acquires characteristics from one or more classes.</td>
</tr>
<tr>
<td>Instantiation</td>
<td>The process of creating an instance of an object, by binding or adding the specific data.</td>
</tr>
<tr>
<td>Message</td>
<td>A request that an object makes to another object to perform an operation.</td>
</tr>
<tr>
<td>Method</td>
<td>An operation upon object, defined as part of the declaration of a class.</td>
</tr>
<tr>
<td>Object</td>
<td>An instantiation of some class which is able to save a state which offers a number of operations to examine or affect its state.</td>
</tr>
<tr>
<td>Operation</td>
<td>An action performed by an object, available to all instances of class and need not be unique.</td>
</tr>
</tbody>
</table>

**Table 1.1 Key Object Oriented Terms for Metrics**
The new object-oriented development methods have their own terminology to reflect the new structural concepts. Referring to Figure 1.7, an object-oriented system starts by defining a class that contains related or similar attributes and operations. The classes are used as the basis for objects. A child class inherits all of the attributes and operations from its parent class, in addition with its own attributes and operations.

The child class can also become a parent class for other classes, forming another branch in the hierarchical tree structure. When an object is created to contain data or information, it is an instantiation of the class. Classes interact or communicate by passing messages. When a message is passed between two classes, the classes are coupled. These specific terms are defined in Table 1.1.

### 1.5.3 Object Oriented Software Development Process

The object-oriented software development process is a sequence of formal activities, supported by specific methodologies and tools. The development process is based on the three classical sequential phases: Object Oriented Analysis (OOA), Object Oriented Design (OOD) and Coding, which is fully iterative and incremental. During Object Oriented Analysis, system requirements are specified first by means of use cases, depicted using Use Case Diagrams and detailed using textual or informal diagrams. Later, the main classes are identified and are grouped into packages. These classes are assigned specific responsibilities and the inheritance hierarchies, dependencies through collaborations and main associations among classes are then identified.

The approach to OOA is fairly compatible with the popular CRC approach (Beck & Cunningham 1989), provided, that all associations among
classes except inheritance are converted to dependencies. The OOD phase starts with detailing all the class responsibilities, which becomes the basic attributes, where the instances of the class must keep track of the specifications of the operations. When the dependencies are detailed or elaborated, which associates itself to aggregation, the structural dependencies are maintained which uses the relationships between the client and the server class. These latter relationships constitute the usage of the instances of the server class and the internal operations of the client class. OOD also includes the detailed specification along with the data structure and the attributes along with the public and private operations of all classes.

The classes can be associated with state diagrams. In all development phases, the system is also described using a dynamic model, showing how the system works during key scenarios. In UML, the dynamic model is described using sequence diagrams, collaboration diagrams and activity diagrams.

Object Oriented Analysis and Design (OOAD) techniques promise many benefits to software developers and software companies, by reusing the software and resilience to change through component libraries and patterns. These benefits include reusability, decomposition of a problem into easily understood objects and the facility of future modifications and enhancements. Despite all of its benefits, the OOAD software development life cycle is by no means more difficult than the typical procedural approach. In fact, it has become even more complicated. Many software engineers are finding it difficult to make the transition into this new way of thinking. In a variety of ways, it is orthogonal to the traditional procedural approach.

Therefore, it is necessary to provide dependable guidelines that one may have to follow to ensure good object-oriented programming practices
and furnish reliable code. One of the most difficult problems in the software engineering realm is producing software that is within budget, complete, on time and that meets the requirements of the end user (Barnes and Swim 1993).

One of the difficulties of evolutionary design is that it is very hard to ensure if design is actually happening. The intermingling of design with programming results in that program is developed without design, where the evolutionary design diverges and fails.

When assessing the design by the quality of the code base, it is found that the code base is getting more complex and difficult to work with and there is no appropriate design being framed on it. Reliable metrics were not available to assess the objective view of Software quality.

1.6 **UNIFIED MODELING LANGUAGE**

There were many more prominent approaches used to capture and accurately express OOD. All these approaches have their advantages. The problem was their dissimilarity. Some approaches explicated required designers to think about the analysis of system as a separate disjoint process from design, while others suggested taking a spiral approach through analysis and design. Finally, Booch, Rumbaugh and Jacobson combined OOD modeling theories and practices to create the Unified Modeling Language (UML). UML is now the industry standard in documenting object-oriented designs. UML provides the ability to capture the characteristics of a system by using a variety of simple and sophisticated notations for documenting systems. These notations are called the nine diagrams of UML (Jacobson et al. 1999).

The more popular diagrams used in UML are the class, sequence and conceptual diagrams. The functionality of UML is too complex for novice
designers to comprehend and hence the designers do not know where to begin when it comes to the designing classes using UML notation. These approaches illustrate the approach or tool used to break a system into simple cohesive units. Breaking the system into cohesive units needs some strategies. The issues to be addressed by the researchers, academics and industry professionals, is how to effectively combine or apply sound OOD approaches to create an industry of proficient Object-oriented designers.

The Unified Modeling Language (UML) (1997) has emerged as a de facto standard for expressing OOA or OOD models. This language is formally described in terms of itself and is based on the types of graphical diagrams. Among these diagrams, the class diagrams and the use case diagrams were suited to be measured using formal metrics. The diagrams exclusively use concepts and entities of UML, although could be also adapted to most other OOA notations. The metrics are intended to allow an early measurement of object-orientedness and quality of the system under development and to make early estimates of its development costs. Moreover, the quality of the system under development can be easily computed if the UML diagrams are stored in computerized form.

The Unified Modeling Language has become the de facto specification standard graphical language for specifying, constructing, visualizing and documenting software systems, business modeling and other non-software systems (Rayner et al. 2005). 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 adopted UML as their organization’s standard. As the amount of UML models produced within an organization increased, the need for measuring their characteristics has also increased in this work.
1.7 RATIONAL ROSE

Rational Rose is a graphical software modeling tool, developed using the Unified Modeling Language as its primary notation. It offers an open API that allows the development of an additional functionality. Rational Rose (the Rose stands for "Rational Object-oriented Software Engineering") is a visual modeling tool for UML. It comes in different versions which suit different requirements. Rational Rose provides support for all the standard features such as UML diagram support, forward and reverse engineering support, documentation and round-trip engineering support. Apart from this, Rational Rose also provides support for version control, IDE integration, design pattern modeling, test script generation and collaborative modeling environment. In addition, Rational Rose also supports the designing of data models within the same environment. An interesting feature of Rational Rose is the ability to publish the UML diagrams as a set of Web pages and images.

With respect to the development of models using the UML, several CASE tools have been developed. Among them, Rational Rose developed by Rational Software is the most prevalent one and is widely used in software development organizations.

1.8 JAVA LANGUAGE EVOLUTION

Java was initially designed by Sun Microsystems (www.java.sun.com) in the early 90s, with its first public release dating back to 1996 (JDK version 1.0). In the first version, only a limited number of standard classes were included (less than 200). Since then, several substantial reorganizations of the language have been undertaken, aimed at two main purposes which are to correct errors and to extend the functionalities of a language. Basically, most of the changes are aimed at improving the development tools, changing standard classes in the API and adding new
packages and classes. For instance, the graphical library Swing, initially included in the releases as an external library, has been later included in the standard API in the version 3 of Java (API 1.3).

Another large-scale modification includes the introduction of inner and nested classes as well as the profound restructuring of the event-handling mechanisms in the version 2 of Java (JDK 1.2). Since the beginning, the portability of the software written in Java has been a major concern and hence it was a key factor in the success of the language. As a matter of fact, Java is the most portable language that is currently used in the software industry. However, other problems, such as lack of downward compatibility between API versions hamper the portability where, for running of the same software on two different platforms, it needs the same version of Java running on both the platforms.

1.9 DESIGN PATTERN

A design pattern is a commonly occurring reusable piece in a software system that provides a certain set of functionality. The identification of a pattern is also based on the context in which it is used. Therefore using patterns in modelling of the systems helps in keeping the design standardized and more importantly, minimizes the reinventing of the wheel in the system design. The class diagram in UML can be used to capture the patterns identified in a system. In addition, UML has a sufficiently extensive and expressive vocabulary to capture the details of patterns. The research focuses only on UML designed object-oriented application and thus provides better support for design pattern issues.
1.10 REVERSE ENGINEERING

Reverse engineering is the process of uncovering the design and the design rationale from a functioning software system. Reverse engineering is an integral part of any successful software system, because changing requirements lead to implementations that drift from their original design. Reverse engineering is an ongoing process in the development of any successful software system as it evolves in response to the changing requirements. The system developers may benefit a lot from reverse engineering techniques that keep track of the design as it drifts from that of earlier versions.

Reverse engineering is a part of the reengineering process in the sense that it focuses on analysing the system and providing representations of a system. The ability to reverse engineer object-oriented legacy systems has become a vital matter in today’s software industry. Early followers of the object-oriented programming paradigm are now facing the problem of transforming their object-oriented legacy systems into frameworks. The need to understand the inner workings of the legacy systems and to identify the potential design anomalies has become important.

“Reengineering is the examination and the alteration of a system to reconstitute it in a new form and the subsequent implementation of the new form”, whereas “Reverse engineering is the process of analysing a system to identify the system’s components and their relationships and to create representations of the system in another form or at a higher level of abstraction” (Chikofsky & Cross 1990).

The problem arises when an effort is made to understand a system, with a preconceived amount of information in mind. The differentiation of the
important pieces of information from the less important information has to be done.

**Metrics:** It is good to use metrics and to measure the system. The number of methods in a class is counted along with the lines of code of a method and the hierarchy nesting level of a class. There is a wide array of possible metrics that can be used and many have been proposed (Lorenz & Kidd 1994). Once the system is measured, assessments can be made about it. The problem is, even with this kind of information, the inner workings of the system cannot be really grasped. The advantage is that assessments can always enrich the semantic properties of a software entity, which can express its size or complexity with a simple number.

**Program Visualization:** An often used approach is to represent design graphically. Such a representation introduces an abstraction level which hides the design behind a graphical entity. Several techniques exist in this domain, which include complex layout algorithms, filtering and interaction. The concept of reverse engineering is mainly about understanding complex things. As the textual representation of source code hinders such understanding in case of a large and complex structure, techniques have been developed to alleviate this problem. All techniques in common must provide an abstraction of the code while at the same time they should cope with scalability and complexity. It is possible that the techniques might work with any amount of code and not break at a certain level of complexity.

**Software Reengineering**

Reengineering a system involves changing large and complex system without the sufficient knowledge of its inner structure and to make the system operable. Software Reengineering can roughly be divided into three
steps, the first two of which have been termed as reverse engineering (Chikofsky & Cross 1990).

**Design Extraction.** Gaining knowledge about the system is an essential factor and can be achieved in many ways:

- Reading the related or existing manuals
- Talking to one of the original developers
- Looking at the source code in a textual or graphical form
- Running the software and observing its behaviour

Once the inner working of the system is understood, it helps to refine the original design decisions which makes the program become the way it is.

**Problem detection.** The design being extracted once, can detect improper design decisions or areas of the system which has to be refactored and redesigned.

**Reengineering.** The system has to be transformed into a state which makes it easier to maintain and to be further developed. Several reengineering tools, techniques and patterns have been developed recently Casais (1998).

### 1.11 MOTIVATION

Most of the existing quality models for object-oriented software development can be applied after product completion or during the final stages of completion. The models rely upon information extracted from the implementation of the product. The information obtained is too late to help in
getting better internal product quality earlier to the completion of the result. Thus, there is a need for models and metrics which can be applied in the early stages of development (requirements and design) to ensure that the analysis and design have favourable internal properties that will lead to the development of a quality end product. This paves a way for the developers to analyse and fix problems, eliminate abnormalities and reduce complexity early in the development cycle.

The application of metrics at the earlier stages will significantly help in reducing rework during and after implementation, in addition to designing better project, effective test plans and resource planning. Fortunately, the object-oriented approach naturally lends itself to an early assessment and evaluation. Object-oriented methodologies need more effort early in the development cycle to identify objects and classes, attributes, operations and relationships. Encapsulation, inheritance and polymorphism require designers to carefully structure the design and to consider the interaction between objects.

The early analysis and design process is an outline for implementation which provides the information needed to assess the software quality, its structure and relationships before it is committed to an implementation.

1.12 PROBLEM SPECIFICATION

Design is a significant factor in the software development process since it controls not only the configuration expense but also the controls the overheads of the phases like implementation and maintenance. Even if the early design is well adequate for the moment, there may be difficulty when trying to change and recover in the maintenance phase. The selection of the
design mainly controls the easiness of maintenance. At least half of the entire life-cycle efforts spend on maintenance, early good design is very cost effective and to maintain high Software quality all the way through the life cycle.

In object-oriented software development the design plays a major part since it fix on the pattern of the software solution. It is difficult to change the design once it has been accomplished Therefore utmost care should be taken for the design from beginning the quality of a design can be assessed and improved with the help of metrics and in object oriented design many metrics have been proposed by various researchers. But most of the metrics are not defined clearly. However, consecutively to automate Software Assessment a precise classification of metrics is needed.

Software Assessment and analysis is needed to make and continue a high worth design with quality assurance. So, in software development it is important and useful to measure Software quality at the earliest. Well-built correlations between design metrics and the maintainability of systems have been recognized as a very important assess for quality assurance.

The concept of Software quality will be defined after the analysis of the measurements by the quality model. Professional assesses the design by using quality model and metrics. However, quality assurance is continual activities in classic designs and are so huge Therefore it is necessary to use automated tools for Software Assessment. Even though a tool cannot replace an individual expert, it can assist us to classify potentially complex design The design alternatives can be evaluated with the help of tool, it assess each alternative and balance the results. An accurate proper description of the metrics will be required for implementing such a tool.
Contribution of the work

The research work proposes an OOP design model which is used to assess any object oriented system.

Organization of the thesis

The First Chapter gives an Introduction on Software Practices and the methodologies used for software development. It also discusses the importance of design in object-oriented system development and the impact of metrics in assessing the quality of design at its early stages. The Second Chapter highlights on the various object-oriented design metrics, tools and the review of literature. The Third Chapter deals with the proposed Software Assessment Model. A Software Assessment Tool is proposed to evaluate the software by applying design metrics. The Fourth Chapter deals with the proposed metrics and the Quality Indicator methodology, which suggests an optimal standard for the major metrics during the Software Assessment. The Fifth Chapter states the need of reverse engineering in software design and also suggests a Reverse Engineering Model to provide Software quality indication for UML based small and medium sized object-oriented projects developed in Java. The Sixth Chapter deals with the Case Study on the quality indications and the imperfections in design found during the early Software Assessment. The Conclusions and the recommendations for future enhancements are summed up in the Seventh Chapter.