Chapter 1

Object Orientation and Software Engineering: A Software Reliability Perspective

1.1 Introduction

Software use is increasing day by day from simple household appliances to research and higher end scientific applications. This has lead to increased reliance on software necessitating the remarkable advances in techniques for software reliability and testing. The most important here is system reliability. System can be decomposed into the hierarchy of related subsystems and components. The reliability of the entire system is related to the reliability of the components by some sort of structure function in which the components may fail independently or in dependent manner. When we consider a system, which consist of hardware, software and human, most often, software is the main source of system reliability problem. Software reliability is the dominant driver of the today’s system reliability [1]. Although hardware reliability is also of much concern there is definite correlation between design techniques and reliability. The major difference between software and other engineering artifacts is that software is developed rather than manufactured. Its unreliability is always the result of design faults, which in turn arises from human intellectual failure [2]. Although advancements have been made towards the production of defect free software, any software required to operate reliably must still undergo extensive testing and debugging. This can be costly and time-consuming process and software analyst require accurate information about how software reliability grows as a result of this process in order to effectively manage their budgets and projects. It therefore requires sound engineering approach for both techniques for producing reliability and sound assessment of the achieved results. Here comes the importance of software engineering.
Software engineering is not only expected to help deliver a software product of required functionality on time and within cost, it is also expected to help satisfy certain quality criteria. The most important one is reliability. Software reliability is the probability of the failure free operation of computer program for a specified period in specified environment. Software reliability engineering can be applied to cover the entire life cycle of the product development from the earliest stage of the product concept through post delivery support of the product. A competitive and mature software development organization targets a high reliability objective from the very beginning of software development.

While applying software reliability engineering to the software, the approach should be taken into account i.e. the software development process observes traditional (structured) approach, object oriented approach or any other. This is essential because, predicting assessing or estimating reliability depend upon the underlying metrics, which are related to the associated properties of quality attributes and the nature of adopted approach.

A surge of new technology, new paradigm new structured analysis concept and new ways of developing software emerged in 1980’s and continues to this date. As we have entered the twenty first century, object oriented paradigm has become widely accepted as a mainstream paradigm for problem solving and software construction. Object oriented technologies enable fast and cost effective application development with reusable component, provides connecting across heterogeneous competing platforms and support integration of data, applications and business process [3]. This and many other factors have led us to concentrate on object-oriented software for software reliability study. While concentrating on object oriented approach object oriented design has been paused primarily.

Object-oriented design plays a pivotal role in software development because it delaines the structure of software solution. Once the design has been implemented, it is difficult and expensive to change. The design phase only takes 5-10% of the total effort (over the whole software life cycle), but a large part (up to 80%) of the total effort goes into correcting bad design decisions [4]. If bad design is not fixed in the design phase, the cost of fixing it after delivery of the software is between 5 and 100 times higher [5].

In order to create and maintain a high quality design, quality assurance in the form of design evaluation and review is needed [4]. The observation show that measures of testing effectiveness and test coverage significantly explains defects. However most of the studies are primarily based on data from software developed using traditional software development methods and our understanding of the applicability of these approaches and metrics in OO development settings is limited [6].

Many of the metrics and quality models currently available for object oriented software analysis can be applied only after a product is complete, or nearly complete. They rely upon information extracted from implementation of the product. This provides information too late to help in improving internal product characteristics prior to the completion of the product [7].
Also the software reliability models available are applicable mostly in testing and operational phase. It is therefore necessary to find metrics and models that can be applied in the early stage of development i.e., particularly in design phase. In object orientation, there is a blurred boundary between object oriented design and analysis, and object oriented design and implementation. Therefore information obtained from the object-oriented design can be used for estimating the defect density. The defect density model can further be used further to predict the reliability of the object oriented software product.

This chapter focuses on software crisis, which is the prime cause for the evolution of software engineering and introduces software reliability perspective in relation to object orientation and software engineering. Chapter 2 produces a study on software reliability, models available, factors and parameters associated with software reliability showing the information processed by most of the models is late in the testing and operational phase and little or no concentration is provided in the design phase. Chapter 3 gives the overall picture of object oriented paradigm and points out the blurred boundaries between analysis, design and implementation, justifying the focus on object oriented design phase for the prediction of software reliability. The existing quality models, metrics, frameworks and methodologies associated with object-oriented design are examined in chapter 4. Chapter 5 produces a new model, framework and set of metrics for predicting the reliability of the object oriented software in the early design phase. Chapter 6 shows details of acquiring the metric values for the set of metrics produced in chapter 5. Chapter 7 shows the result and analysis of the software reliability prediction.

Throughout the research several tools were used like CASRE tool (appendix A) is used for showing the trend test and model ranking process for the s1 data set (see appendix C). Functions from statistical toolbox of MALAB (appendix A) are used for performing statistical analysis. For mathematical computation use of Excel is preferred. A new tool called RFQMOOD (Reliability Focused Quality Model for Object Oriented Design) tool is developed which provides a complete process from extracting the object oriented design metrics to reliability estimation (appendix E).

### 1.2 Software Crisis:

The phrase *software crisis* alludes to a set of problems encountered in the development of computer software during the 1960s. Within this period the software industry unsuccessfully attempted to build larger and larger software systems (to match advances in hardware technology) by simply scaling up existing development techniques. As a result:

1. Schedule and cost estimates were often grossly inaccurate.
2. Productivity of programmers could not keep up with demand.
3. Poor quality and less reliable software was produced.
4. Always changing approaches towards development.
5. Inability to keep pace with the changing technology.
6. Human efficiency at intellectual level and, egoistic problems and many more.

The lack of progress in solving software development problems was designated the "software crisis." This term was coined with respect to DoD software projects. Early efforts to address the "software crisis" by the DoD in particular included the following:

1. Software Standards: The DOD-STD-2167 development standard defined the "waterfall" development life cycle, consisting of requirements analysis, design, code/unit test, component test, and system test. Each life cycle phase had entry and exit criteria and the phases were sequential.
2. Modern Programming Languages: The DoD mandated the development and use of the Ada Language for Mission Critical Systems (1985). The use of this modern programming language was intended specifically to address the maintenance problems caused by the proliferation of processors and languages. Ada was developed to standardize software development environments and to address programming methods for real-time embedded systems. Ada introduced the concept of language support for software engineering practices such as data abstraction and encapsulation.

By the late 1980s, it was clear that the structure of the development process imposed by DOD-STD-2167 was not the solution to the software crisis. In addition, object-oriented methods and languages were not compatible with the standards. Even though DOD-STD-2167 was revised, the "waterfall" life cycle, which involves a sequential flow of activities, was found impossible to sustain. This was due to uncertainty and dynamic changes in the functional requirements of most large systems. Feedback between the waterfall phases of DOD-STD-2167 was cumbersome and time consuming.

To find solutions to the "software crisis," the Advanced Research Development Agency (ARPA) of the US Government funded the Software Engineering Institute (SEI) at Carnegie Mellon University in 1984. The SEI formalized the software process into a Capability Maturity Model (CMM) framework. This process identifies procedures and activities such as project planning and tracking, configuration management, requirements management, and training. These activities guide the development of high-quality, reliable software.

The Capability Maturity Model (CMM) figure 1.1 describes the principles and practices underlying software process maturity and is intended to help organizations improve the maturity of their software processes through an evolutionary path. CMM describes five levels of maturity. It is a scale against which projects and organizations can be classified, representing their capability to develop and deliver software products with predictable characteristics. Maturity Levels provide a layer in the foundation for continuous process improvement. As organizations establish and improve the software processes, they progress through different levels of maturity.
On the basis of the SEI's CMM, an organization performs a self-assessment in cooperation with an SEI-licensed vendor. An organization's current development process is evaluated against the CMM Key Process Areas and the assessment provides the organization with the actions that need to be taken to improve its development process. The CMM measures a software development organization's ability to develop software products reliably, on time and within budget. Other recent software process models, such as the European ISO-9000 standards are similar to the CMM.

ISO-9000 provides guidelines for an organization's software quality management system. Companies are ISO-9000 certified via an evaluation process similar to the SEI Software Process Assessments. Both the CMM and ISO-9000 require documented software development procedures. The most widely used standards are listed in table 1.1. Most of these standards are applicable to both traditional and object oriented softwares. Also Object Management Group (OMG) - develops standards to help make object applications to be portable and communicate between each other (interoperability). They have developed the Component Object Request Broker Architecture (CORBA) standard along with object and OODBMS interfaces.

### Table 1.1 Software Standards

<table>
<thead>
<tr>
<th>Organization</th>
<th>Standard</th>
<th>Purpose of standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>U.S Department of Defense (DoD)</td>
<td>1. DoD – STD – 2167A</td>
<td>Defense system software development</td>
</tr>
<tr>
<td></td>
<td>2. MIL – STD – 498</td>
<td>Software development and documentation</td>
</tr>
<tr>
<td></td>
<td>3. MIL – STD – 882C</td>
<td>System safety program requirements</td>
</tr>
</tbody>
</table>
Along with these standards other process standards such as Personal Software Process (PSP) and Team Software Process (TSP) are also observed. PSP is a disciplined approach that an individual software developer can use to improve the quality and efficiency of his or her personal work. TSP describes how teams of software engineers can work together effectively. However, to effectively handle software development procedure one should know software engineering.

### 1.3 Software Engineering

To address these problems in section 1.1 the discipline of software engineering came into being. The term software engineering is credited to Fredrick Bauer of the University of Munich [8]. He and his colleagues first used the term software engineering...
as a descriptor for a 1968 NATO – sponsored workshop and had significant impact in making visible the technical and economic aspects of an emerging software discipline. Several authors and committees tried to define software engineering. Here is the definition offered by IEEE computer society [9]. Software Engineering:

(1) The application of a systematic, disciplined approach to the development, operation and maintenance of software, that is, the application of engineering to software.
(2) The study of approaches as in (1).

Software is the major part of computer systems, and hence the field of software engineering can expect similar rapid growth. The principal forces affecting software engineering today include [10]:

1. Greater and increasingly international business competition,
2. Higher cost of both information system development failures.
3. Faster pace of change in computing technology, and
4. Increasing complexity of the managing information system development.

All software needs high reliability (with failure being either inconvenient or deadly) and needs to be produced economically. These needs are addressed by software engineering. Software engineering techniques improve the functionality and efficiency of applications and the ease and efficiency of software developers.

There have been debates about whether software engineering is “real” engineering and whether software development is an art, a science, or an engineering discipline [11]. To justify software engineering as engineering discipline consider software engineering as a set of activities [13] as explained in next section.

1.3.1 Software Engineering: A Set of Activities

The various activities are:

◊ Modeling Activities
◊ Problem Driven Activities
◊ Knowledge Acquisition Activities
◊ Rational Driven Activities

A. Modeling Activities

A model is an abstract representation of a system that enables us to answer questions about the system. Models are useful when dealing with systems that are too large, too small, too complicated or too expensive to experience first hand. Models allow us to visualize and understand system that no longer exist or that are only claimed to exist. The two basic challenges faced by the software engineers are:
◊ They need to understand the system environment in which the system has to operate.
◊ They need to understand the system they could build, to evaluate different solutions and trade offs.

Modeling helps software engineers to deal with complexity, by focusing at any one time on only the relevant details and ignoring everything else. Object oriented methods combine the application domain and solution domain modeling activities into one. The application domain is first modeled as a set of objects and relationships. This model is then used by the system to represent the real world concepts it manipulates. Then, solution domain concepts are also modeled as objects. The idea of object oriented methods is that the solution domain model is the transformation of application domain model.

**B. Problem Driven Activities**

Engineering is a problem solving activity. The engineering methods include five steps:

◊ Formulate the problem
◊ Analyze the problem
◊ Search the solution
◊ Decide on appropriate solutions
◊ Specify the solutions

To fit these engineering methods in object oriented software development process, let us see the development activities which make up the software engineering development process. They are requirement elicitation, analysis, system design, object Design and implementation.

During requirement elicitation and analysis, software engineers formulate the problems with the client and build the application domain model. Requirement elicitation and analysis corresponds to step 1 and 2 of engineering methods. During system design, software engineers analyze the problem; break it into smaller pieces, and select general strategies for designing the system. During object design, they select detail solution for each piece and decide on most appropriate solution. System design and object design result in solution domain model. System and object design correspond to step 3 and 4 of engineering methods. During implementation, software engineers realize the system by translating the solution domain model into an executable representation. Implementation corresponds to step 5 of engineering method. What makes software engineering different from problem solving in other sciences is that change occurs in the application and solution domain while the problem is being solved.

**C. Knowledge Acquisition Activities**

Software engineering is knowledge engineering activity. In modeling the application and solution domain, software engineers collect data, organize it into information and
formalize it into knowledge. Knowledge acquisition is not sequential, as a single piece of additional data can invalidate complete models.

**D. Rational Driven Activities**

Software engineering is a rational driven activity. When acquiring knowledge and making decisions about the system or it’s application domain, software engineers also need to capture the context in which decisions were made and the rational behind these decisions. Rational information represented as a set of issue models, enables software engineers to understand the implication of the proposed change when revisiting the decision.

Thus it is found that software engineering concentrates on various activities. Every activity has some role to play in various software engineering phases. Software engineering phases can be viewed as traditional software engineering and as object-oriented software engineering. Section 1.2.2 briefs the phases in traditional software engineering and section 1.4 briefs object oriented paradigm.

**1.3.2 Phase in Traditional Software Engineering**

Figure 1.2 shows different phase of software engineering in general. The individual phases of the above life cycle are briefly described below:

![Figure 1.2 Phases Associated With A General Model Of The Software Life Cycle.](image)

**A. Requirements analysis and specification.**

Work begins by establishing and analyzing the requirements of the procurer (usually the end user), i.e. determining the nature of the system to be built (in terms of hardware and software). Requirements analysis usually results in a requirements specification document or (for small software systems) a requirements specification statement. This
specification then defines the "product", and thus must be accepted by both the customer and the developers. In other words it has some legal standing.

**B. System Analysis and Design.**

The second phase is to analyze the requirements and produce a "high-level" design of the proposed system (again in terms of hardware and software). There are many approaches to system analysis and design.

**C. Software Design.**

The third phase is to produce a detailed design of the desired software sometimes referred to as a software specification. The idea behind a software design/specification is that it is (implementation) "language independent". You should be able to give the design to different programmers to implement in different languages and get a working result in all cases. The detailed design can be thought of as a "model" of the system that is envisaged, and its implementation its physical realization.

**D. Unit Testing and Implementation.**

During implementation the design is translated into a machine-readable (and consequently executable) form. Some "unit" testing normally takes place as the implementation progresses. By unit testing we mean the testing of sub-assemblies of the program.

**E. Testing**

Once the implementation has been completed it must be tested to ensure that all parts of the program behave in the manner that they are expected to behave. One straightforward approach in which software can be tested is to identify "meaningful" test cases with which the software can be run. The results produced can then be compared with the expected results and appropriate conclusions drawn. On completion of testing the product is delivered to the customer.

**F. Maintenance**

As with any other commercial product, software must be maintained once it has been delivered. We can identify three categories of maintenance:

- Corrective Maintenance - Correction of errors that were not discovered earlier on in the development process.
- Adaptive maintenance - modification to accommodate changes in the software's external operating environment.
- Perfective maintenance - inclusion of additions functionality not originally identified during requirements analysis and specification, i.e, extending the software beyond its original requirements.
Maintenance, although perceived as being unglamorous, takes up a considerable amount of the total effort expanded throughout the software life cycle. It is difficult to quantify the amount of effort involve, but some authors claim between 50% and 70% of the total effort depending on the nature of the application. A primary aim of program development should therefore be to reduce the effort required for maintenance by designing the code with a view to "easy maintenance”. This brings the need of re-engineering.

1.4 The Need for Re-engineering

The increasing reliance on information technology for consumer and industrial goods imposes new requirements on software flexibility. The major trends in customer requirements are customer specific modifications and software versions (custom made systems), much faster response to change requests and new requirements (evolution), and the ability to easily modify the software based on the immediate customer needs (tailoring).

Object oriented programming has often been promoted as the most effective approach to build inherently flexible software, and was quickly adopted by industry in the recent years. There are already applications consisting of millions of lines of code developed in several hundred man-years. While the benefits of object-oriented technology are widely recognized, its utilization does not necessarily result in general, adaptable families of systems. These large applications are often suffering from improper use of object-oriented techniques (like inheritance) and the lack of object oriented methods being geared towards the construction of families of systems instead of building single applications. The result is a new generation of inflexible legacy systems.

In order to better meet customer requirements, the industrial users need to re-engineer these monolithic object oriented legacy systems to flexible frameworks and libraries of small, understandable software components. Such frameworks will allow a greater flexibility to varying needs of different categories of customers, as well as an easier integration of new requirements:

- a large pool of stable components can be easily mixed and matched together to reconfigure standard functionality for different categories of customers;
- customer specific components can be easily added to the framework;
- the scope of changes to tailor a system to specific customer needs can be confined to small parts of the framework, generally to a single component;
- because the systems are no longer monolithic, completely new functionality can be implemented by extensions to frameworks and do not necessarily entail a complete redesign or a completely separate version of the system;
- components can be replaced by commercially available software if needed.

Object oriented paradigm promotes re-engineering and various object oriented metrics can be identified that are suitable from the perspective of re-engineering.
1.5 Object Oriented Paradigm

The history of science let it be social or technological reveals the truth of change from one era to another. The software industry though one of the youngest industry has undergone tremendous changes since last 50 years. Thus one of the fundamental challenges the software industry is dealing with is change. The need to develop easy to extend and change software system has driven interest in new approaches to software development and design [12]. The software development process has undergone changes from traditional approach to object oriented approaches in almost every phase that is analysis, design, programming, testing, metrics and measurements and bench marking.

The key features of object orientation such as encapsulation, inheritance, polymorphism with dynamic binding, object identity and the Liskov type substitution principle has motivated for the adoption of object oriented paradigm to many software developers and therefore to this study. Object-oriented paradigm is both a programming style (Object-Oriented Programming --- OOP) as well as an approach to software engineering. The principal features of the object orientation are flexibility and adaptability, software reuse and extension, information hiding consistent notation and integration of the various software engineering phases and reliability.

Compared to traditional software development phase the phases in object oriented software development in general would be as shown in figure 1.3.

![Figure 1.3 Refined Software Engineering Life Cycle Model for the Object Oriented Paradigm](image)
1.6 Software Reliability

The main motive behind applying software engineering norms and or adopting object orientation is to produce quality product. There are many different models for software quality, but in almost all models, reliability is one of the criteria, attribute or characteristic that is incorporated. ISO 9126 defines six quality characteristics, one of which is reliability. Building high reliability software depends on the application of quality attributes at each phase of the development life cycle with the emphasis on error prevention, especially in the early life cycle phases. IEEE Std 982.2-1988 includes the diagram in Figure 1.4, indicating the relationship of reliability to the different life cycle phases[14].

Most users of software products cite software reliability as the most important feature of the software products they use. Hence it is crucial that software reliability engineering techniques should play a central role in the planning and control of software development projects. In particular, it is important to document the times and nature of bug occurrences, and their correction times, throughout the design and implementation phases as well as during the formal testing phase. With such data it is possible to estimate suitable reliability which can be used to estimate the time at which the software product will reach a target level of reliability, or to devise methods to decrease that time. Various software reliability models have been proposed in literature. However these models are applicable in the late development phase. Design is the area were one can focus for improving software reliability to handle future crisis.

![Figure 1.4 Relationship of Reliability to the Different Life Cycle Phases](image-url)
Software development process is an intellectual activity when supported by software engineering norms and object orientation can produce quality software. Reliability being one of the important attribute of quality if traced from the early phase of software development can be more useful. This therefore necessitates concentrating on the various aspects related to software reliability for predicting the reliability of the software product at the object oriented design phase.

References: