Chapter 1

Introduction

1.1 Introduction

With the advent of the digital computers, human society depends on the computing activities in its day-to-day life. The computer revolution is fuelled by rapid technological advancement. Computer hardware and software permeates our modern society. Computers are embedded in mobile phones, wrist watches, home appliances, automobiles and in almost all devices we use in our daily life. The size and complexity of computer-intensive systems has grown dramatically during the past decade and the trend is continuing. For example, in the Telecommunications Industry, operations for phone carriers are supported by hundreds of software systems with hundreds of millions of lines of source code. Hardware has sufficient technological advancement whereas software industries are not in a position to match the demand. The demand for complex software systems has increased rapidly than the ability to design, implement, test and maintain them. When the requirements for and dependencies on computers increase, the possibility of crises of computer failures also increases. The impact of these failures ranges from inconvenience e.g. malfunctioning of the device to economic damage, interruption of banking systems, loss of life, failure of flight system and medical software. Needless to say, the reliability of computer systems has become a major concern for our society. The study focuses on software and its reliability. Furthermore, the study examines reliability which means software reliability of product.

In recent years, the cost of developing software and penalty cost of software failure is a matter of concern in a system. Failure of the software may result in an unintended system state of course of action. Locating software faults is extremely difficult and costly. A study conducted by Microsoft showed that it takes about twelve programming hours to locate and correct a software defect. This shows the gravity of software failure and its impact on the human society.
Software failures have led to serious consequences in business also. According to media reports software problem contributed to a recent Delhi metro system in early 2010. The software reportedly failed to perform as expected in detecting and preventing excess power usage. A review board also concluded that the NASA Mars Polar Lander failed in December 1999 due to software problems that caused improper functioning of retro rockets utilized by the Lander as it entered the Martian atmosphere. Software design error and insufficient software testing caused an explosion that grounded the maiden flight of the European Space Agency's (ESA) Ariane 5 rocket, less than seconds after take-off on 4th June 1996.

Before specifying the actual problem domain, to acquire background knowledge some basic concepts related to software development are discussed below. The chapter functions as a blue print of the entire study.

1.2 Software Engineering

Software plays a central role in almost all aspects of daily life: in government, banking and finance, education, transportation, entertainment, medicine, agriculture and law. The number, size and application domains of computer programs have grown dramatically; to meet out the needs of software a standard approach came into existence as a separate engineering discipline i.e. software engineering. Software engineering is an application of science, mathematics and a systematic approach of software development by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation. Software engineering has to deal with a different set of problems than other engineering disciplines, since the nature of software is different. A software product is entirely a conceptual entity. Due to its conceptual nature, there is an "intellectual distance" between the software and the problem the software is solving. This intellectual distance makes it hard for a person, to understand the problem and the software used to solve the problem. Due to the this distance, person responsible for maintaining the software has to spend considerable effort in trying to understand the software that solves the problem. This makes the software maintenance very expensive. Due to this it is necessary to have a clearly defined process for developing the software. As a result, hundreds of billions are being spent on software
development, and the livelihood and lives of most people depend on the effectiveness of this development. Software products have helped us to be more efficient and productive. They make us more effective and efficient problem solvers, and they also provide us an environment for work and play that is often safer, more flexible and less confining. Despite these successes, there are serious problems in the cost, time and quality of many software products and therefore time to time demands for a new, efficient and effective technology for software development necessitate looking and relooking into different software developmental approaches. object-oriented approach and the traditional/procedural approach. Till date object-based technique is used most frequently and suitable to conceptualize the real world activities.

1.3 Software Development Process

In the 1960s and 1970s software industries were facing crises and its development projects were characterized by massive cost over runs and schedule delays; the focus was on planning and control. The emergence of Waterfall process to help and tackle the growing complexity of development projects was a logical event (Boehm, B., 1976). A simplified Waterfall model includes various phases of software development is described in the Figure 1.1 below.

Fig. 1.1 Phases of Software Development
**System Requirements Phase:** In this phase very high level functionality/specifications of the product is defined and typically includes feasibility analysis. The domain experts will do this and assess what is feasible for this project.

**Software Requirement Analysis:** Once the feasibility is established a detailed analysis is done in order to understand the problem which the software system is to solve. The functionality is explored in detail and documented in this phase. All necessary system functional and non-functional requirements are detailed in this phase. Systems analysts will be performing this activity.

**Software Design:** Design is the earliest phase in which the system structure is clearly defined. It is further subdivided into small phases, System/architectural design and detailed/component design. System design aims to identify the modules that should be in the system, the specifications of these modules and how they interact with each other and to external applications to produce the required results. Once overall system design is ready individual project groups parallel start working on the high level design followed by low level design. In object-oriented approach classes are considered as components. Therefore detailed/component design is a class level design. Broadly, system design defines what is to be achieved and detailed design defines how it is achieved.

**Coding/Implementation:** In this phase the detailed designs are implemented using the language specified in the requirements. The goal of the coding phase is to translate the design in a given programming language. The coding phase effects both testing and maintenance. A well written code can reduce the testing and maintenance effort.

**Software Integration and formal System Test:** Once the coding is done testing is performed. Testing is the major quality control measure employed during the software development. Its basic function is to detect faults.

**Operation and Maintenance:** During this phase, the system is installed and put to use after necessary training to the users. This phase continues until the system is operating in production in accordance with the defined user requirements. Any changes requested or faults found will be fixed during the maintenance stage.
The IEEE standard dictionary measures to produce reliable software classifies the software development process into early, middle, and late segments, as shown in Table 1.1.

**Table 1.1 Software Development Process Classification**

<table>
<thead>
<tr>
<th>Development process segments</th>
<th>Phases</th>
<th>Reliability Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early</td>
<td>Requirements specifications, analysis and Design</td>
<td>The early segment relates to the potential causes of system reliability</td>
</tr>
<tr>
<td>Middle</td>
<td>Implementation, Testing, Installation and checkout</td>
<td>The middle segment relates to the reduction of process errors that can improve the efficiency of software development.</td>
</tr>
<tr>
<td>Late</td>
<td>Operations, Maintenance and Retirement</td>
<td>The late segment relates to actual system performance reliability.</td>
</tr>
</tbody>
</table>

In software development process design is very important phase where the system is designed according to the specifications. In object-oriented approach design specifications are implemented without any change. Thus design phase is a earliest phase, which provides static information about the structural properties of product to be implemented into reality during its development. This is very much clear from the above classification also that design phase can act as early feedback provider to produce reliable software. Therefore this research also focuses on class level design phase to introduce design complexity measures to identify high risk components/classes. The basic constructs of object-oriented design at class level are given below:
**Cohesion**: Cohesion refers to the internal consistency within the parts of the design. Cohesion is centered on data that is encapsulated within an object and on how methods interact with data to provide well-bound behaviour. A class is cohesive when its parts are highly correlated. It should be difficult to split a cohesive class. Cohesion can be used to identify the poorly designed classes. Booch definition can further clarify, “Cohesion measures the degree of connectivity among the elements of a single class or object” (Booch, 1994).

**Coupling**: Coupling indicates the relationship or interdependency between modules. For example, object X is coupled to object Y if and only if X sends a message to Y that means the number of collaboration between classes or the number of messages passed between objects. Coupling is a measure of interconnecting among modules in a software structure.

**Inheritance**: Inheritance is a mechanism whereby one object acquires characteristics from one or more other objects. Inheritance occurs in all levels of a class hierarchy. “Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship” (Rumbangh et al., 1991)

In general, conventional software does not support this characteristic because it is a pivotal characteristic in many object-oriented systems as well as many object-oriented metrics focus on it.

**Encapsulation**: Encapsulation is a mechanism to realize data abstraction and information hiding. Encapsulation hides internal specification of an object and shows only the external interface. “The process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation” (Booch, 1994).

Encapsulation influences metrics by changing the focus of measurement from a single module to a package of data.

**Information Hiding**: Booch (1994) states that, information hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics. An object has a public interface and a private representation; these two elements are kept
distinct. Information hiding acts a direct role in such metrics as object coupling and the
degree of information hiding. “All information about a module should be private to the
module unless it is specifically declared public”, (Meyer, B, 1998)

Localization: In object oriented design approach localization is based on objects. In a
design, if there is some changes in the localization approach, the total plan will be
violated, because one function may involve several objects, and one object may provide
many functions. So localization is the process of gathering and placing things in close
physical proximity to each other.

Metrics should apply to the class as a complete entity. Even the relationship between
functions and classes is not necessarily one-to-one. For that reason, “metrics that reflect
the manner in which classes collaborate must be capable of accommodating one-to-many
and many-to-one relationships” (Pressman, 2001).

1.4 Software Metrics

Software metrics is a broad term for all those activities which involves some
degree of software measurement and are intended to measure the software quality and
performance characteristics quantitatively, encountered during the successive phases of
software development. These can serve as measures of software products for the purpose
of comparison, cost estimation, fault prediction and forecasting. Metrics can also be used
in guiding decisions throughout the life cycle, determining whether software quality
improvement initiatives are financially worthwhile.

A lot of research has been conducted on software metrics and their applications.
Different set of software metrics are suitable for individual development approaches to
measure the internal and external attributes of process, products and resources.

1.4.1 The Need for Software Metrics

Software development historically has been the arena of the artist. Artistically
developed code often resulted in arcane algorithms or spaghetti code that was
unintelligible to those who had to perform testing and maintenance. Initially only very
primitive measures such as lines of code (LOC) and development time per stage of the
development life cycle were collected. Projects often exceeded estimated time and budget. In the pursuit of greater productivity, software development evolved into software engineering. Part of the software engineering concept is the idea that the product should be controllable. DeMarco reminds us that what is not measured cannot be controlled.¹

Measurement is the process whereby numbers or symbols are assigned to dimensions of entities in such a manner as to describe the dimension in a meaningful way. An entity may be a thing or an event, i.e., a person, a play, a developed program or the development process. A dimension is a trait of the entity, such as the height of a person, the cost of a play. Obviously, the entity and the dimension to be measured must be specified in advance. Sandro Morasca provides a list of usefulness of measurement as it permeates everyday life and is an essential part in every scientific and engineering discipline. Measurement allows the acquisition of information that can be used for developing theories and models, and devising, assessing, and using methods and techniques. Practical application of engineering in the industry would not have been possible without measurement.²

Measurements cannot be taken and then applied to just any dimensions. Unfortunately this is exactly what the software development community has been doing, (Fenton, 1994) e.g., lines-of-code, being a valid measurement of size, has been used to "measure" the complexity of programs.

An intuitive and empirical assessment of the entities and dimensions must be preserved by the measurement (the assignment of numbers and symbols). For example, when measuring the height of two people the taller person should be assigned a larger number than the shorter person. Notice that the unit of measure (feet, inches, meters) has no effect on this rule.

Because people observe things differently (and often intuitively feel differently about things), a model is usually defined for the entities and dimensions to be measured. The model requires everyone to look at the subject from the same viewpoint. Fenton uses the example of human height. Should posture be taken into consideration when...
measuring human height? Should shoes be allowed? Should the measurement be made to
the top of the head or the top of the hair? The model forces a reasonable consensus upon
the measurers. 3

There are two types of measurement; direct measurement of a dimension requires
only that dimension; indirect measurement of a dimension requires that one or more other
dimensions be measured. Because the dimensions of greatest interest, e.g., quality and
reliability, are often external to the entity being measured and therefore is very hard to
measure directly, indirect measurement usually achieves more useful results. That is,
internal dimensions are measured, e.g., verbal skills, to assess external dimensions, e.g.,
IQ. As has already been stated, control of a process or product requires that the process or
product is measurable; therefore, control of software requires software measures (Baker
et al., 1990). It does no good to measure the process if the product is not measured. Being
the best at producing an inferior product does not define a quality process.

Many researchers have asserted the importance of software measurement. Vollman
wrote of the importance of software measurement to society while Grady and Caswell
and Chidamber and Kemerer described its importance to management. 4, 5, 6 Walrad and
Moss contended that 1) effective development policies require clear definitions and
explicit measures, 2) measurement compels quality, and 3) well-defined quality is
measurable. 7 Barnes and Swim noted that the objectives of a project must be clearly
stated in measurable terms. Most scientific theories express relations among quantities.
To test a theory or to apply it therefore requires measurement. 8 Kyburg asserts that in
order to keep the software process in control, you have to institute a policy of careful and
effective measurement. 9 Thus it reasons that all measures are indeed metrics. In
particular, a measure becomes a metrics when it has been demonstrated that it is a valid
representation of the dimension being measured. In this work the term metric and
measure are used interchangeably.

1.4.2 Software Metrics and Measurement Theory

Measurement theory was first used in software metric research to validate the
myriad complexity metrics which dominated the early research in the field. Correlations
were expected to exist between the complexity of a project and the achievement of acceptable parameters in its development. This was the rationale for the interest in software complexity and the development of measures to measure this complexity (Anderson, 1992). Measurement theory helps us to define and validate metrics of specific complexity dimensions. (Fenton, 1994)

When measuring source documents (analysis, design, or code), it must be demonstrated that an improvement in the measurement, however it is achieved, corresponds to an improvement in the document. This is especially true if a measurement is used to evaluate an employee's work. However, it should not be possible to improve the measurement through incidental or trivial modifications of the document. If money and time are to be spent to improve the measurement, the measure must truly represent the property being measured. (Kearney, et al., 1986)

When defining a measure first, one must designate precisely the dimension to be measured, e.g., the height of humans. Then a model is specified that captures the dimension, e.g., stand up straight, take off your shoes, do not include hair height in the measurement. The congruence that comes from the model must represent the dimension being measured, i.e., the intuitive order of the entities, with respect to the dimension being measured, must be preserved by the model. Finally, an order-preserving map from the model to a number system is defined, e.g., if it is observed that Harry is taller than Dick, any measurement taken of their height must result in numbers or symbols that preserve this relationship. (Baker, et al., 1990)

Before a numerical mapping (or even a model) can be proposed, it must be known what is being measured. This basic measurement principle has been ignored in much of the software metric work of record. It is fundamental to measurement theory that the measurer has an intuitive understanding, usually based on observation, of the dimension being measured. (Fenton, 1991)
1.4.3 Classification

Different design and implementation techniques are used in software development process depending upon their suitability and efficiency. Two commonly used techniques are traditional and object-oriented. These techniques require different set of metrics. Therefore, the software metrics are classified into two categories. Every software development project domain, whatever design approach used includes basic entities as process, product and resources. To measure attributes of these entities process, product and resource metrics are required. Classification of software metrics is shown in the figure given below:

![Software Metrics Classification Diagram](image)

**Fig. 1.2 Software Metrics Classification**

1.4.3.1 Traditional Metrics

The standard components of any software system are data and procedures. Historically most programming languages used for business have been procedural languages. Procedural languages emphasize the procedures to be performed on the data. Languages are integral and crucial to the solution in a procedural paradigm. In a procedural paradigm a program is built around the control structures of sequence,
iteration, and selection. Traditional software complexity metrics are used to measure the control flow, data flow and data structure in a program. Control flow addresses the sequence in which instructions are executed, data flow follows the trail of a data item as it is created or handled and data structure is the organization of data itself within the program (Fenton & Pfleeger, 1998). Complexity-based metrics are intended to evaluate the difficulty to understand, develop and maintain the software. The cyclomatic complexity counts the number of possible logic paths in a piece of code (McCabe, 1976). It considers the possible flows that an execution can follow that impact in the code readability and maintainability, as well in the number of tests cases for a good coverage. The Halstead effort (Halsted, 1977) is a metric that considers the number of operands and operators to evaluate the effort and the volume of written code. Coupling metrics evaluate how intensive and how dispersed is the coupling in the system. The coupling intensity can be measured by using the number of operation calls per operation, and the coupling dispersion can be measured as the number of called classes per operation call. A more recent approach for the coupling measurement, called dynamic coupling, counts the number of calls at runtime and not statically (Fenton et al., 1994). Source lines of code and comment percentage are some well known traditional size metrics.

1.4.3.2 Object-oriented Metrics

Object-oriented development models objects based on real-world concepts. This paradigm stresses the encapsulation of data and procedure, thus, de-emphasizing the procedure. Encapsulation makes changes less likely to impact the rest of the program. More than one procedural routine, known as methods in standard object-oriented language, can be stored with the data in a single class. A method is the instantiation of an operation for a class (Smith, 1991); there is a hierarchy of object definitions. Object is discrete and distinguishable entity. Classes are groups of objects that share the same attributes (data) and operations (behaviour). A class usually relates to some real-world concept, i.e., vehicles. This class would include all traits relative to all vehicles. Further specialization would take place within subclasses, i.e., petrol-vehicles and diesel-vehicles. Further detail is relegated to still lower subclasses, i.e., two-wheelers and four-
wheelers. Each lower level of subclass inherits the traits of all super classes that the subclass is related to (Smith, 1991). The basic principles and concept used in two approaches are totally different. The traditional metrics are only applicable at method level to quantify the various entities of an object-oriented software.

It is necessary that software engineers have quantitative measurements for object oriented software projects, especially for the design phase. These measures allow designers to access the software early in the process, making changes that will reduce complexity and improve the continuing capability of the product. A significant number of object-oriented metrics have been developed in literature. For example, metrics proposed by Chidamber and Kemerer, 1994; Li and Henry, 1993; MOOD metrics (Abreu & Fernando, 1995); Lorenz and Kidd, 1994 etc. C.K. metrics are the most popular (used) among them to measure class complexity at design level. Another comprehensive set of metrics is MOOD metrics. Essentially any object-oriented metrics is an attempt to measure or predict some internal or external attribute of product, process, or resource. These are designated as process metrics, product metrics and resource metrics.

**Internal attributes**

Internal attributes of a product, process or resource are those that can be measured purely in terms of the product, process, or resource itself. In other words, an internal attribute can be measured by examining the product, process or resource on its own.

**External attributes**

External attributes of a product, process or resource are those that can be measured only with respect to how the produce process or resource, relates to its environment. Here, the behaviour of the process, product or resource is important, rather than the entity itself.

*Table 1.2* provides the summary of internal and external attributes of process, product and resources of software development project.

**Table 1.2**
Table 1.2 Components of Software Measurements (Fenton & Pfleeger, 1998, p76)

<table>
<thead>
<tr>
<th>Entities</th>
<th>Attributes</th>
<th>Internal</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Products</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specification</td>
<td>Size, reuse, modularity, redundancy, functionality, syntactic correctness.</td>
<td></td>
<td>Comprehensibility, maintainability,</td>
</tr>
<tr>
<td>Design</td>
<td>Size, reuse, modularity, coupling, cohesiveness, inheritance, functionality.</td>
<td></td>
<td>Quality, complexity, maintainability.</td>
</tr>
<tr>
<td>Test data</td>
<td>Size, coverage level.</td>
<td></td>
<td>Quality, reusability.</td>
</tr>
<tr>
<td><strong>Processes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constructing Specification</td>
<td>Time, effort, number of requirements changes.</td>
<td></td>
<td>Quality, cost, stability.</td>
</tr>
<tr>
<td>Detailed design</td>
<td>Time effort, number of specification faults found.</td>
<td></td>
<td>Cost, cost-effectiveness.</td>
</tr>
<tr>
<td>Testing</td>
<td>Time, effort, number of coding faults found.</td>
<td></td>
<td>Cost, cost-effectiveness, stability, ...</td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personnel</td>
<td>Age, price.</td>
<td></td>
<td>Productivity, experience, intelligence.</td>
</tr>
<tr>
<td>Teams</td>
<td>Size, communication level, structuredness.</td>
<td></td>
<td>Productivity, quality.</td>
</tr>
<tr>
<td>Organization</td>
<td>Size, ISO Certification, CMM level</td>
<td></td>
<td>Maturity, profitability.</td>
</tr>
<tr>
<td>Software</td>
<td>price, size.</td>
<td></td>
<td>Usability, reliability.</td>
</tr>
<tr>
<td>Hardware</td>
<td>Price, speed, memory size.</td>
<td></td>
<td>Reliability.</td>
</tr>
</tbody>
</table>
1.4.3.2.1 Process Metrics

Process metrics are measures of the process used to obtain the product. Processes are set of software related activities which are used to measure the status and progress of the system design and to predict future effects. A process is usually related to a timescale. The timing can be explicit, as when an activity must be finished by a specific date, or implicit, as when one activity must be finished before another can begin. The object oriented process metrics are intended to measure the followings (Jacobson et al., 1992).

- Total development time,
- Development time in each process and sub process,
- Time spent to modify models from previous processes,
- Time spent in all kinds of sub process, such as use case specification, object specification, use case design, block design, block testing and use case testing for each particular object,
- Number of different kind of fault found during reviews,
- Number of change proposals on previous models,
- Cost for quality assurance,
- Cost for introducing new development process and tools.

1.4.3.2.2 Products Metrics

Product metrics are measures of product at any phase of software development from requirements to system delivered. Product metrics are used to control the quality of the software product. These metrics are applied to incomplete software products in order to measure their complexity and to predict properties of the final product. Products are any artifacts, deliverables or documents that result from a process activity. Products are not restricted to the items that management is committed to deliver to the customer. Any
artifact or document produced during the software life cycle can be measured. The object oriented product metrics are intended to measure as follows (Jacobson et al., 1992)

- Number, width and height of the inheritances hierarchies,
- Number of classes inheriting a specific operation,
- Number of classes that a specific class is dependent on,
- Number of classes that are dependent on a specific class,
- Number of direct users of a class or operation.

1.4.3.2.3 Resource Metrics

Resource metrics are measure of the behaviour and the development environment. Resources are entities required by a process activity. The resources that we want to measure include any input for software production. Thus, personnel, materials, tools, and methods are candidates for measurement through resource metrics and some of them are as under:

- Programmer efficiency in terms of program size
- Performance efficiency of software as a tool

1.5 Software Reliability

Software reliability is defined as the probability of failure-free software operations for a specified period of time in a specified environment.

Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However software reliability is different from hardware reliability in the sense that software does not wear out, burn out or deteriorate i.e reliability does not decrease with time. Moreover software generally enjoys reliability growth as long as the software is not modified.

Software Reliability is an important attribute of software quality, together with functionality, usability, performance, serviceability, capability, installability,
maintainability and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. For example, large next-generation aircraft will have over one million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming international Space station will have over two million lines on-board and over ten million lines of ground support software; several major life-critical defense systems will have over five million source lines of software. While the complexity of software is inversely related to software reliability, it is directly related to other important factors in software quality, especially functionality, capability, etc. Emphasizing these features will tend to add more complexity to software. These are four commonly used methods to acquire software reliability of system and are discussed below in subsection.

1.5.1 Fault Prevention

Fault Prevention deals with preventing faults being incorporated into a system. This can be accomplished by use of development methodologies and good implementation techniques.

1.5.2 Fault Removal

Fault Removal can be sub-divided into two sub-categories: i) removal during development and ii) removal during use. Removal during development requires verification so that faults can be detected and removed before a system is put into production. Once systems have been put into production a system is needed to record failures and remove them via maintenance cycle.

1.5.3 Fault Tolerance

Software fault tolerance is a necessary part of a system with high reliability. It is a way of handling unknown and unpredictable software (and hardware) failures (faults) by providing a set of functionally equivalent software modules developed by diverse and
independent production teams. The assumption is the design diversity of software, which itself is difficult to achieve.

1.5.4 Fault Forecasting

Fault Forecasting predicts likely induced faults in classes so that they can be removed or their effects can be circumvented through redesign rigorous testing and reviews. Faults are predicted in terms of structural properties of software products, measured by various complexity metrics in early phases of software development.

1.6 Statement of the Problem and its Significance

The need to make good design decisions as early as possible in the software development process as possible is critical to the success or failure of many development efforts. Bad design decisions found after coding can cost as much as 6.5 times the cost if found during design, 60 to 100 times if not found until production (Pressman, 1992). This makes it imperative that there be a quantitative way to make good design decisions early in the development process. The identification, proposition and verification of useful design metrics is needed in order to move the evaluation of design decisions from subjective to objective. This evaluation of early design options will allow for the better identification of problems before coding.

Many proponents of object technology proclaim it as the most promising software technology to date. Object technology is the latest in a series of development approaches that will irrevocably transform the software engineering process. It promises to alleviate cost, schedule, and quality problems commonly found in software development (Chidamber, 1994). Closing the gap between technology and business, object technology brings the possibility of delivering quality products to the market place more quickly and at a lower cost. This technology also promises to bring the power of business object modeling and process reengineering to software development improving the usefulness of the software products along the way (Henderson-Sellers, 1996). All of these promises of object technology have combined to make it an increasingly popular choice for
software development. Software metrics evaluated at design phase play a vital role in
keeping all promises of object technology.

1.7 Originality of the Study

Many object-oriented metrics are being proposed. It is not clear that these metrics
are valid measures of the dimensions that they claim to measure because they have not
been validated using measurement theory. Some of these metrics are touted as predictive
without being rigorously defined. This study looks at new object-oriented design metrics
and their validity.

Does the metric measure, what its author proposes to measure? If not, what can be
said about the metric in terms of what is being measured? Is there a measure (a validated
metric) which does measure the desired dimension? Are the statistics used with the metric
appropriate, considering the scale attributed to the metric? Is the measurement an
assessment measurement or meant to be a predictive measurement? Does the metric hold
up under vigorous scrutiny of the conditions of representation and uniqueness? Do
intuitive and empirical understandings survive under all allowable transformations?

The sole force behind this work woke up from the desire to improve the quality of
object-oriented designs. The quality of work done early in the software development
process is very dependent on the level of expertise the designers have. The need is critical
for a method of identifying errors and problem areas early in development cycle. The
design process needs to move from an unsound or shaky to a sound engineering practice
using quantitative analysis and measurement to compare designs and design choices.
Design errors have a significant impact on the cost of developing software and can be the
cause of complete failure if found too late in the development process. There has been
some work on metrics for object-oriented design. What is lacking is the information
needed to make the measurement of designs practical before coding phase starts.

The focus of this research is object-oriented design metrics. Although it is found
that a few existing metrics measure the complexity at the design level and even don't
clarify all the object-oriented design concepts. Therefore we proposed a new metric suite
to measure the complexity of class at design phase as an early predictor of fault contents to acquire reliability. These metrics measures the calls complexity in terms of some design concept of object technology; data hiding, inheritance, aggression (coupling) and cohesion. The scholar is hopeful that the analysis will narrow the field of metrics into a subset that can be the focus for further study and verification.

1.8 Objectives of the Study

On the basis of scope and limitations of the study, the present study aims to achieve the following objectives:

(i) to examine the existing object-oriented software metrics for further use in predicing the faults in a class that affect product reliability;

(ii) to illustrate the need of new measures in identifying the errors and problem areas early in the software development process;

(iii) to propose a new metrics suite to find out class complexity at design level in an object oriented system;

(iv) to validate the proposed design metrics as early predictor of fault contents to enhance reliability.

1.9 Scope and Limitations of the Study

The work provides the broad aspect of the areas i.e., object-oriented metrics. In the scope, the study proposes to examine a large number of metrics for quantifying the complexity of class at design level. Association of these metrics with external quality attributes of the products is explored through real world data. The metrics used in the study are well validated. While limiting the study it covers only metrics at the design levels to provide early feedback. The design metrics will be the thrust area of investigation and the faults are attempted to be predicted to acquire reliability. It will not investigate in detail the reliability aspect, as it may require another full-fledged study. However the study is an academic investigation for a doctorate degree.
1.10 Hypotheses Formulated

A hypothesis is a tentative statement that proposes a possible explanation to some phenomenon or event. A useful hypothesis is a testable statement which may include a prediction. A hypothesis is a specific, testable prediction about what you expect to happen in your study. "A hypothesis is a logical supposition, a reasonable guess, an educated conjecture. It provides a tentative explanation for a phenomenon under investigation" (Leedy & Ormrod, 2001). Hypothesis is a formal statement that presents the expected relationship between an independent and dependent variable. Hypotheses always offer explanations for the relationships between those variables that can be empirically tested. Hypotheses are also important because they help an investigator to locate information needed to resolve the research problem or sub problems (Leedy & Ormrod, 2001). For the present study, the following research questions have been postulated:

H1 Internal metrics have significant relationship with external attributes of product;

H2 Proposed metrics predict faults at the design level of the object-oriented software;

H3 There is a significant relationship among external links and induced faults in a class.

1.11 Scheme of Work

A scheme of work in this study is employed to show the methodology including the design, analysis and presentation of the work.

1.11.1 Methodology

The methodology section answers two main questions; (i) How was the data collected or generated? and (ii) How was it analyzed? In other words, it shows the reader how, one obtained his/her results. But why do one need to explain how one obtained his/her results? One needs to know how the data was obtained because the method affects the results. Knowing how the data was collected helps the reader to evaluate the validity
and reliability of one’s results and the conclusions one draws from them. Often, there are
different methods that the researcher can use to investigate a research problem.
Methodology should make clear the reasons as to why the researcher chose a particular
method or procedure. The research methods must be appropriate to suit to the objectives
of the study.

1.11.1.1 Analytical Validation of the Metrics

Consistent with the desire to move metrics research into a more rigorous footing,
it is desirable to have formal sets of criteria with which to evaluate proposed metrics.
This research uses two successive frameworks given by Weyuker and Braind to
analytically evaluate the four proposed design metrics. Weyuker has developed a formal
list of properties for software metrics and has evaluated a number of existing software
metrics using these properties (Weyuker, 1988). This list, while still a subject of debate
and refinement (Cherniavsky & Smith, 1991), is a widely known formal analytical
approach used by other OO metrics researchers (Chidamber & Kemerer, 1994). Briand
has developed a formal list of properties for complexity metrics which was widely used
by the researchers to analyse various types of complexity metrics (Briand, 1996).

1.11.1.2 Empirical Validation of the Metrics

Empirical validation of metrics explore the application area with the help of real-
world data. This is also used to check the hypotheses and theoretical view point of
cognitive theory about the association between independent and dependent variable. This
research empirically validates the design metrics using data from real-world projects.
Design metrics are used as independent variables and number of faults as dependent
variable (Linear regression analysis for predications). The empirical validation not only
results in validated metrics, but provides relationship between design metrics and
number-of-faults at class level, to predict the number of fault and identifying the classes
at high-risk. Therefore software reliability can be achieved by focusing on the high-risk
components for locating and removal of faults may cause failure.
Design metrics as early measures are appropriate to reduce overall effort of development work and to achieve reliability of software. This work also aims to introduce the design metrics to acquire some level of product reliability before the coding phase starts using the fault forecasting techniques. For the sake of convenience and clarity, the entire spectrum of the present investigation is presented in the following seven broad chapters.

**Chapter 1 Introduction**

This chapter functions as the blueprint of the entire work. It begins with an introductory note followed by statement of the problem, significance of the study, objectives, scope and limitation, hypotheses of the study and scheme of the work to be carried out.

**Chapter 2 Review of Related Literature**

This chapter is completely devoted to review of previous literature related to the present study to acquire the knowledge of past and to show the current trends in the field. The review covers six broad subject domains involved with the present research problem.

**Chapter 3 Research Design and Methodology**

For any investigation proper methodology is very much important. Designing a method of investigation is the brain of the whole work. This is also true here in this case. This chapter provides the overview of whole research process and methods/techniques used at various stages.

**Chapter 4 Fault Prediction Analysis**

In chapter fourth the most popular existing metrics Weighted Methods per Class (WMC), Response for a Class (RFC), Coupling between Objects (CBO), Depth of Inheritance (DIT), Number of Children (NOC), Lack of Cohesion of Methods (LCOM) and Source Lines of Code (SLOC) in software industries and research organizations used for
measurement in object-oriented environment are empirically analysed to predict the faults at class-level in early stages.

Chapter 5 Towards Newly Proposed Design Complexity Metrics

This chapter introduces four new design metrics namely CMCM (Class Member Complexity measure), CICM (Class Inheritance Complexity Measure), CALM (Class Aggregation Level Measure) and CCOM (Class Cohesion Measure) to measure the class-complexity induced by the various design concept of object-technology i.e. data-hiding, inheritance, aggregation and cohesion. Cognitive theory is taken as theoretical basis for the development of design metrics.

Chapter 6 Empirical Validation of Proposed Metrics and Reliability

Chapter six focuses on software reliability, induced faults and reliability, fault-forecasting technique to acquire reliability, analysis of design metrics using cognitive theory to observe association with induced faults and empirical analysis is done to authenticate the theoretical basis of metrics development depicted by cognitive analysis results and to find the level and mode of association along with to develop best faults prediction model. Next to these, faults prediction model is further validated through spearman's correlation coefficient and its utility is described to acquire some level of product reliability in early phases of development (before the coding starts) through redesign by identifying classes having more faults.

Chapter 7 Conclusion, Findings and Areas for Further Research

The last chapter is the concluding part of entire work. In this chapter the result is presented in the form of findings, hypotheses testing. On the basis of the result the trend setting pattern is known.

Bibliography

Technical speaking this is not a chapter. However, the scholar feels this bibliography is not less than any chapter. Someone says good research is nothing but the haunting for good literature and presenting those in the form of bibliography. An
extensive list of related literature is provided here just to claim that the literature search was conducted extensively. The list is organized with the help of Publication Manual of American Psychological Association (APA) 5th ed. Washington, D.C.: American Psychological Association. 2002

1.13 Conclusion

Chapter I of the work entitled "Introduction" functions as the blueprint of the entire work. It begins with an introductory note followed by the logical successive link of the work. The importance of the object-oriented design is highlighted along with the background information of software engineering. The basic concept of object-oriented software design approach is to save cost, manpower and even severe consequences of software disaster. The thrust area of the Ph.D work software metrics is highlighted. In this chapter the basic research questions are mentioned in the form of research hypotheses. A comprehensive list of objectives of the work is also propagated which can provide sense of the work in brief. The chapter further includes the originality and significance of the work. The chapter briefly speaks of the entire work.

Notes

2. See Chap. 26 (Software measurement) by Sandro Morasca, a course metarial prescribed by University of Pittsburg.

25


References


