CONTENTS

SOFTWARE ENGINEERING

SYLLABUS

UNIT I

Introduction: Software Crisis - Software Myths - Software Life Cycle Models: Build and Fix Model Waterfall Model - Prototyping Model - Iterative Enhancement Model - Evolutionary Development Model - Spiral Model - Capability Maturity Model - ISO 9000, 9001 and 9002, Software Metrics.

UNIT II

Software Project Planning: Cost Estimation - The constructive Cost Model - The Putnam Resource Allocation Model - Software Risk Management - Software Requirement Analysis and Specification. Requirements Engineering - Problem Analysis - Approaches - Software Requirements Specification (SRS) - Behavioural Requirements - Non-behavioural Requirements.

UNIT III

Software Design: Conceptual and Technical Designs - Modularity - Dependence Matrix - Strategy of Design: Bottom-up Design - Top-down Design - Hybrid Design - Function-oriented Design - Object-oriented Design.

UNIT IV

Software Reliability: Importance - Software reliability - Hardware reliability - Failures and Faults - Reliability Concept - Reliability Models - Reliability Allocation. Software Testing: Testing Process - Functional Testing - Structural Testing - Test activities - Debugging - Testing Tools.

UNIT V

Software Maintenance: Categories of Maintenance - Problems during maintenance - solutions - the maintenance process - Maintenance Models - Reverse Engineering - Software Re-engineering - Estimation of Maintenance costs - Configuration Management - Documentation: User Documentation - System Documentation - Classification Schemes.

UNIT I

LESSON

1

SOFTWARE ENGINEERING: AN INTRODUCTION

CONTENTS

1.0 AIMS AND OBJECTIVE

After studying this lesson, you should be able to:

- Explain the evolving role of the software
- Explain software crisis
- Describe software relating myths
- Define software metrics

1.1 INTRODUCTION

The complexity and nature of software have changed tremendously in the last four decades. In the 70s applications ran on a single processor, received single line inputs and produced alphanumeric results. However, the applications today are far more complex running on client-server technology and have a user friendly GUI. They run on multiple processors with different OS and even on different geographical machines.

The software groups work hard as they can to keep abreast of the rapidly changing new technologies and cope with the developmental issues and backlogs. Even the Software Engineering Institute (SEI) advises to improve upon the developmental process. The "change" is an inevitable need of the hour. However, it often leads to conflicts between the groups of people who embrace change and those who strictly stick to the traditional ways of working.

Thus, there is an urgent need to adopt software engineering concepts, practices, strategies to avoid conflicts and in order to improve the software development to deliver good quality software within budget and time.

Evolution of the Software Role

The role of software has undergone drastic change in the last few decades. These improvements range through hardware, computing architecture, memory, storage capacity and a wide range of unusual input and output conditions. All these significant improvements have lead to the development of more complex and sophisticated computer-based systems. Sophistication leads to better results but can cause problems for those who build these systems.

Lone programmer has been replaced by a team of software experts. These experts focus on individual parts of technology in order to deliver a complex application. However, the experts still face the same questions as that by a lone programmer:

- Why does it take long to finish software?
- Why are the costs of development so high?
- Why aren't all the errors discovered before delivering the software to customers?
- Why is it difficult to measure the progress of developing software?

All these questions and many more have lead to the manifestation of the concern about software and the manner in which it is developed – a concern which lead to the evolution of the software engineering practices.

1.2 SOFTWARE CRISIS

It has been expected that, by 1990, the one half of work force will depend on computers and software to do its daily work. As computer hardware costs persist to decline, the demand for new applications software continues to boost at a rapid rate. The existing stock of software continues to grow, and the effort required for maintaining the stock continues to increase as well. At the same time, there is a major shortage of qualified software professionals. Combining these factors, one might project that at some point of time, every worker will have to be concerned for software development and maintenance. For now, the software development scene is often characterized by:

- *Size:* Software is becoming larger and more complex with the growing complexity and expectations out of software. E.g. the code in consumer products is doubling every couple of years.
- *Quality:* Many software products have poor quality. I.e., the software produces defects after put into use due to ineffective testing techniques. E.g. Software testing typically finds 25 defects per 1000 lines of code.

- *Cost:* Software development is costly i.e., in terms of time taken to develop and the money involved. E.g. Development of the FAA's Advance Automation System cost over $700 per line of code.
- *Delayed delivery:* Serious schedule overruns are common. Very often the software takes longer than the estimated time to develop which in turn leads to cost shooting up. E.g. one in four large-scale development projects is never completed.

1.3 SOFTWARE MYTHS

- *Myth 1:* Computers are more reliable than the devices they have replaced. Considering the re-usability of the software, it can undoubtedly be said that the software does not fail. However, certain areas which have been mechanized have now become prone to software errors as they were prone to human errors while they were manually performed. Example Accounts in the business houses.
- *Myth 2:* Software is easy to change. Yes, changes are easy to make - but hard to make without introducing errors. With every change the entire system must be re-verified.
- *Myth 3:* If software development gets behind scheduled, adding more programmers will put the development back on track. Software development is not a mechanistic process like manufacturing. In the words of Brooks: "adding people to a late software project makes it later".
- *Myth 4:* Testing software removes all the errors. Testing ensures presence of errors and not absence. Our objective is to design test cases such that maximum number of errors is reported.
- *Myth 5:* Software with more features is better software. This is exactly the opposite of the truth. The best programs are those that do one kind of work.
- *Myth 6:* Aim has now shifted to develop working programs. The aim now is to deliver good quality and efficient programs rather than just delivering working programs. Programs delivered with high quality are maintainable. The list is unending. These myths together with poor quality, increasing cost and delay in the software delivery have lead to the emergence of software engineering as a discipline.

1.4 SOFTWARE METRICS

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality.

The software crisis must be addressed and, to the extent possible, resolved. To do so requires more accurate schedule and cost estimates, better quality products, and higher productivity. All these can be achieved through more effective software management, which, in turn, can be facilitated by the improved use of software metrics. Current software management is ineffective because software development is extremely complex, and has only few well-defined, reliable measures of either the process or the product to guide and evaluate development. Thus, accurate and effective estimating, planning, and control are nearly impossible to achieve. Improvement of the management process depends upon improved ability to identify, measure, and control essential parameters of the

development process. This is the goal of software metrics—the identification and measurement of the essential parameters that affect software development.

Software metrics and models have been proposed and used for some time. Metrics, however, have rarely been used in any regular, methodical fashion. Recent results indicate that the conscientious implementation and application of a software metrics program can help achieve better management results, both in the short run (for a given project) and in the long run (improving productivity on future projects).

1.4.1 Definition of Software Metrics

It is important to further define the term *software metrics*. Essentially, *software metrics* deals with the measurement of the software product and the process by which it is developed. The software product should be viewed as an abstract object that evolves from an initial statement of need to a finished software system, including source and object code and the various forms of documentation produced during development. Ordinarily, these measurements of the software process and product are studied and developed for use in modeling the software development process. These metrics and models are then used to estimate/predict product costs and schedules and to measure productivity and product quality. Information gained from the metrics and the model can then be used in the management and control of the development process, leading, one hopes, to improved results.

Good metrics should facilitate the development of models that are capable of predicting process or product parameters, not just describing them. Thus, ideal metrics should be:

- Simple, precisely definable—so that it is clear how the metric can be evaluated;
- Objective, to the greatest extent possible;
- Easily obtainable (*i.e.*, at reasonable cost);
- Valid—the metric should measure what it intended to measure; and
- Robust—relatively insensitive to (intuitively) insignificant changes in the process or product.

In addition, for maximum utility in analytic studies and statistical analyses, metrics should have data values that belong to appropriate measurement scales. It has been observed that the fundamental qualities required of any technical system are:

- Functionality—correctness, reliability, etc.;
- Performance—response time, throughput, speed, etc.; and
- Economy—cost effectiveness.

1.4.2 Classification of Software Metrics

Software metrics may be broadly classified as either *product metrics* or *process metrics*. *Product metrics* are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program (either source or object code), or the number of pages of documentation produced. *Process metrics*, on the other hand, are measures of the software development process, such as overall development time, type of methodology used, or the average level of experience of the programming staff.

In addition to the distinction between product and process metrics, software metrics can be classified in other ways. One may distinguish *objective* from *subjective* properties (metrics). Generally speaking, objective metrics should always result in identical values for a given metric, as measured by two or more qualified observers. For subjective metrics, even qualified observers may measure different values for a given metric, since their subjective judgment is involved in arriving at the measured value. For product metrics, the size of the product measured in Lines of Code (LOC) is an objective measure, for which any informed observer, working from the same definition of LOC, should obtain the same measured value for a given program. An example of a subjective product metric is the classification of the software as "organic," "semi-detached," or "embedded," as required in the COCOMO cost estimation model. Although most programs might be easy to classify, those on the borderline between categories might reasonably be classified in different ways by different knowledgeable observers. For process metrics, development time is an example of an objective measure, and level of programmer experience is likely to be a subjective measure.

Another way in which metrics can be categorized is as *primitive* metrics or *computed* metrics. Primitive metrics are those that can be directly observed, such as the program size (in LOC), number of defects observed in unit testing, or total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some manner from other metrics. Examples of computed metrics are those commonly used for productivity, such as LOC produced per person-month (LOC/person-month), or for product quality, such as the number of defects per thousand lines of code (defects/KLOC). Computed metrics are combinations of other metric values and thus are often more valuable in understanding or evaluating the software process than are simple metrics.

Software Measurement Theory

Software Measurement Theory is the techniques or methods that apply software measures to software engineering objects to achieve predefined goals. A measure is a mapping from a set of software engineering objects to a set of mathematical objects. Measurement goals vary with the software engineering object being measured, the purpose of measurement, who is interested in these measurements, which properties are being measured, and the environment in which measurement is being performed. Examples of measures include software size, Halstead's software science measures, and McCabe's cyclomatic complexity. Associated models include sizing models, cost models, and software reliability models.

Measurement Scales for Software Metrics

Software metric data should be collected with a specific purpose in mind. Ordinarily, the purpose is for use in some process model, and this may involve using the data in other calculations or subjecting them to statistical analyses. Before data are collected and used, it is important to consider the type of information involved. Four basic types of measured data are recognized by statisticians are—nominal, ordinal, interval, and ratio. The four basic types of data are described by the following table:

Table 1.1: Classification of Software Metrics

Type of Data	Possible Operations	Description of Data
Nominal	$=, \neq$	Categories
Ordinal	$<, >$	Rankings
Interval	$+, -$	Differences
Ratio	/	Absolute zero

Operations in this table for a given data type also apply to all data types appearing below it.

Examples of software metrics can be found for each type of data. As an example of nominal data, one can measure the type of program being produced by placing it in to a category of some kind—database program, operating system, etc. For such data, one cannot perform arithmetic operations of any type or even rank the possible values in any "natural order." The only possible operation is to determine whether program A is of the same type as program B. Such data are said to have a nominal scale, and the particular example given can be an important parameter in a model of the software development process. The data might be considered either subjective or objective, depending upon whether the rules for classification allow equally qualified observers to arrive at different classifications for a given program.

Ordinal data, by contrast, allow us to rank the various data values, although differences or ratios between values are not meaningful. For example, programmer experience level may be measured as slow, medium, or high. (In order for this to be an objective metric, one must assume that the criteria for placement in the various categories are well defined, so that different observers always assign the same value to any given programmer.)

Data from an interval scale cannot only be ranked, but also can exhibit meaningful differences between values. McCabe's complexity measure [McCabe76] might be interpreted as having an interval scale. Differences appear to be meaningful; but there is no absolute zero, and ratios of values are not necessarily meaningful. For example, a program with complexity value of 6 is 4 units more complex than a program with complexity of 2, but it is probably not meaningful to say that the first program is three times as complex as the second.

Some data values are associated with a *ratio* scale, which possesses an absolute zero and allows meaningful ratios to be calculated. An example is program size, in lines of code (LOC). A program of 2,000 lines can reasonably be interpreted as being twice as large as a program of 1,000 lines, and programs can obviously have zero length according to this measure. It is important to be aware of what measurement scale is associated with a given metric. Many proposed metrics have values from an interval, ordinal, or even nominal scale. If the metric values are to be used in mathematical equations designed to represent a model of the software process, metrics associated with a ratio scale may be preferred, since ratio scale data allow most mathematical operations to be meaningfully applied. However, it seems clear that the values of many parameters essential to the software development process cannot be associated with a ratio scale, given our present state of knowledge. This is seen, for example, in the categories of COCOMO.

1. *Product Metrics:* Most of the initial work in product metrics dealt within the characteristics of source code. A number of product metrics are discussed below.
    - *Size Metrics:* A number of metrics attempt to quantify software "size." The metric that is most widely used, LOC, suffers from the obvious deficiency that its value cannot be measured until after the coding process has been completed. Function points and system *Bang* have the advantage of being measurable earlier in the development process—at least as early as the design phase, and possibly earlier.
        - *Lines of Code:* A line of code (or LOC) is possibly the most widely used metric for program size. It would seem to be easily and precisely definable; however, there are a number of different definitions for the number of lines of code in a particular program. These differences involve treatment of blank lines and comment lines, non-executable

statements, multiple statements per line, and multiple lines per statement, as well as the question of how to count reused lines of code. The most common definition of LOC seems to count any line that is not a blank or comment line, regardless of the number of statements per line. LOC has been theorized to be useful as a predictor of program complexity, total development effort, and programmer performance (debugging, productivity).

- *Function Points:* Albrecht has proposed a measure of software size that can be determined early in the development process. The approach is to to compute the total Function Points (FP) value for the project, based upon the number of external user inputs, inquiries, outputs, and master files. The value of FP is the total of these individual values, with the following weights applied: inputs: 4, outputs: 5, inquiries: 4, and master files: 10. Each FP contributor can also be adjusted within a range of ± 35% for specific project complexity. Function points are intended to be a measure of program size and, thus, effort required for development.
- *Bang:* DeMarco defines system Bang as a function metric, indicative of the size of the system. In effect, it measures the total functionality of the software system delivered to the user. Bang can be calculated from certain algorithm and data primitives available from a set of formal specifications for the software. The model provides different formulas and criteria for distinguishing between complex algorithmic versus heavily data oriented systems. Since Bang measures the functionality delivered to the user, DeMarco suggests that a reasonable project goal is to maximize "Bang per Buck"—Bang divided by the total project cost.

❖ ***Complexity Metrics:*** Numerous metrics have been proposed for measuring program complexity—probably more than for any other program characteristic. The examples discussed below are some of the better-known complexity metrics. As noted for size metrics, measures of complexity that can be computed early in the software development cycle will be of greater value in managing the software process. Theoretically, McCabe's measure [McCabe76] is based on the final form of the computer code. However, if the detailed design is specified in a program design language (PDL), it should be possible to compute v(G) from that detailed design.

- *Cyclomatic Complexity—v(G):* Given any computer program, one can draw its control flow graph, G, wherein each node corresponds to a block of sequential code and each arc corresponds to a branch or decision point in the program. The cyclomatic complexity of such a graph can be computed by a simple formula from graph theory, as $v(G) = e - n + 2$, where e is the number of edges, and n is the number of nodes in the graph. McCabe proposed that v(G) can be used as a measure of program complexity command, hence, as a guide to program development and testing. For structured programs, v(G) can be computed without reference to the program flow graph by using only the number of decision points in the program text [McCabe76]. McCabe's cyclomatic complexity metric has been related to programming effort, debugging performance, and maintenance effort.
- *Extensions to v (G):* Its been noted that McCabe's cyclomatic complexity measure, v(G), provides a measure of program complexity but fails to differentiate the complexity of some rather simple cases involving single conditions (as opposed to multiple conditions)

in conditional statements. As an improvement to the original formula, extending v(G) to v •(G) = [l:u], where l and u are lower and upper bounds, respectively, for the complexity.

- *Knots:* The concept of program knots is related to drawing the program control flow graph with a node for every statement or block of sequential statements. A knot is then defined as a necessary crossing of directional lines in the graph. The same phenomenon can also be observed by simply drawing transfer-of-control lines from statement to statement in a program listing. The number of knots in a program has been proposed as a measure of program complexity.
- *Information Flow:* The information flow within a program structure may also be used as a metric for program complexity. Basically, the method counts the number of local information flows entering (fan-in) and exiting (fan-out) each procedure. The procedure's complexity is then defined as:

  $c = [\text{procedure length}] \times [\text{fan-in} \times \text{fan-out}]^2$

2. *Process Metrics:* Software metrics may be defined without specific reference to a well-defined model, as, for example, the metric LOC for program size. However, more often metrics are defined or used in conjunction with a particular model of the software development process. In this text, the intent is to focus on those metrics that can best be used in models to predict, plan, and control software development, thereby improving our ability to manage the process. Effective models allow us to ignore uninteresting details and concentrate on essential aspects of the artifact described by the model. Preference should be given to the simplest model that provides adequate descriptive capability and some measure of intuitive acceptability. A good model should possess predictive capabilities, rather than being merely descriptive or explanatory.

   In general, models may be analytic-constructive or empirical-descriptive in nature. There have been few analytic models of the software process, the most notable exception being Halstead's software science, which has received mixed reactions. Most proposed software models have resulted from a combination of intuition about the basic form of relationships and the use of empirical data to determine the specific quantities involved.

   Ultimately, the validity of software metrics and models must be established by demonstrated agreement with empirical or experimental data. This requires careful attention to taking measurements and analyzing data. In general, the work of analyzing and validating software metrics and models requires both sound statistical methods and sound experimental designs. Precise definitions of the metrics involved and the procedures for collecting the data are essential for meaningful results. Small-scale experiments should be designed carefully, using well-established principles of experimental design. Unfortunately, validation of process models involving larger projects must utilize whatever data can be collected. Carefully controlled large experiments are virtually impossible to conduct.

   A knowledge of basic statistical theory is essential for conducting meaningful experiments and analyzing the resulting data. In attempting to validate the relationships of a given model, one must use appropriate statistical procedures and be careful to interpret the results objectively. Most studies of software metrics have used some form of statistical correlation, often without proper regard for the theoretical basis or limitations of the methods used. In practice, software engineers

lacking significant background in statistical methods should consider enlisting the aid of a statistical consultant if serious metric evaluation work is undertaken.

Representative examples of software models are presented below.

- ***Empirical Models:*** This is one of the earliest models used to project the cost of large-scale software projects. The method relates a proposed project to similar projects for which historical cost data are available. It is assumed that the cost of the new project can be projected using this historical data. The method assumes that a waterfall-style life cycle model is used. A (25 × 7) structural forecast matrix is used to allocate resources to various phases of the life cycle. In order to determine actual software costs, each software module is first classified as belonging to one of six basic types—control, I/O, etc. Then, a level of difficulty is assigned by categorizing the module as new or old and as easy, soft, medium, or hard. This gives a total of six levels of module difficulty. Finally, the size of the module is estimated, and the system cost is determined from historical cost data for software with similar size, type, and difficulty ratings.
- ***Statistical Models:*** The metric LOC was assumed to be the principal determiner of development effort. A relationship of the form

$$E = aL^b$$

was assumed, where L is the number of lines of Uncode, in thousands, and E is the total effort required, in person-months. Regression analysis was used to find appropriate values of parameters a and b. The resulting equation was

$$E = 5.2\,L^{0.91}$$

Nominal programming productivity, in LOC per person-month, can then be calculated as L/E. In order to account for deviations from the derived form for E, a productivity index, I was developed, which would increase or decrease the productivity, depending upon the nature of the project. The computation of I was to be based upon evaluations of 29 project variables.

1.4.3 The GQM Approach

The Goal Question Metric (GQM) approach is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals. Thus it is important to make clear, at least in general terms, what informational needs the organization has, so that these needs for information can be quantified whenever possible, and the quantified information can be analyzed a to whether or not the goals are achieved.

The approach was originally defined for evaluating defects for a set of projects in the NASA Goddard Space Flight Center environment. The application involved a set of case study experiments and was expanded to include various types of experimental approaches. Although the approach was originally used to define and evaluate goals for a particular project in a particular environment, its use has been expanded to a larger context. It is used as the goal setting step in an evolutionary quality improvement paradigm tailored for a software development organization, the Quality Improvement Paradigm,

within an organizational framework, the Experience Factory, dedicated to building software competencies and supplying them to projects.

According to many studies made on the application of metrics and models in industrial environments, measurement in order to be effective must be:

1. Focused on specific goals;
2. Applied to all life-cycle products, processes and resources; and
3. Interpreted based on characterization and understanding of the organizational context, environment and goals.

This means that measurement must be defined in a top-down fashion. It must be focused, based on goals and models. A bottom-up approach will not work because there are many observable characteristics in software (e.g., time, number of defects, complexity, lines of code, severity of failures, effort, productivity, defect density), but which metrics one uses and how one interprets them it is not clear without the appropriate models and goals to define the context.

There are a variety of mechanisms for defining measurable goals that have appeared in the literature: the Quality Function Deployment (QFD) approach, the Goal Question Metric (GQM) approach, and the Software Quality Metrics approach. This section will present the GQM approach and provide examples of its application.

The Goal Question Metric (GQM) approach is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals. Thus it is important to make clear, at least in general terms, what informational needs the organization has, so that these needs for information can be quantified whenever possible, and the quantified information can be analyzed as to whether or not the goals are achieved. Although the approach was originally used to define and evaluate goals for a particular project in a particular environment, its use has been expanded to a larger context. It is used as the goal setting step in an evolutionary quality improvement paradigm tailored for a software development organization. The result of the application of the Goal Question Metric approach application is the specification of a measurement system targeting a particular set of issues and a set of rules for the interpretation of the measurement data. The resulting measurement model has three levels:

- *Conceptual level (GOAL):* A goal is defined for an object, for a variety of reasons, with respect to various models of quality, from various points of view, relative to a particular environment. Objects of measurement are:
  - *Products:* Artifacts, deliverables and documents that are produced during the system life cycle; e.g., specifications, designs, programs, test suites.
  - *Processes:* Software related activities normally associated with time; e.g., specifying, designing, testing, interviewing.
  - *Resources:* Items used by processes in order to produce their outputs; e.g., personnel, hardware, software, office space.
- *Operational level (QUESTION):* A set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterizing

model. Questions try to characterize the object of measurement (product, process, and resource) with respect to a selected quality issue and to determine its quality from the selected viewpoint.

- *Quantitative level (METRIC):* A set of data is associated with every question in order to answer it in a quantitative way. The data can be:
  - *Objective:* If they depend only on the object that is being measured and not on the viewpoint from which they are taken; e.g., number of versions of a document, staff hours spent on a task, size of a program.
  - *Subjective:* If they depend on both the object that is being measured and the viewpoint from which they are taken; e.g., readability of a text, level of user satisfaction.

Given the importance of defining a measurement goal, the following text provides a template structure, which divides a goal into three components as follows:

Purpose

- Measurement approach e.g. to understand, classify, compare
- Object, either a product (e.g. functional requirements) or a process (e.g. design inspection)
- Reason e.g. improve, eliminate

Perspective e.g. software engineer, customer, project manager.

Environment

A description of the organization in which the measurement object is embedded. To illustrate use of the template to structure measurement goals, consider the following, informally stated measurement goal.

To improve the accuracy of software project managers' cost estimates at the specification stage of a project within the Telecommunication Systems Division at X organization.

This can be restructured to yield:

- *Purpose*
  - Understand
  - Factors that influence project costs
  - To improve costs estimation accuracy at the specification stage
- *Perspective:* Project manager.
- *Environment:* The Telecommunication Systems Division at X organization.

The measurement approach will be to characterize project cost estimates so as to determine which factors lead to accurate and inaccurate estimation. This information can then form the basis of a program to improve estimation process at the Telecommunication Systems Division at X organization. On occasions it can be quite difficult to specify the measurement approach at this early stage, consequently, there can be a tendency to retreat into the use of "weasel" words such as "analyze" and "assess". Still, it is always possible to return to a goal and refine it during the measurement process.

It is generally accepted that progressing from goal to question is the most difficult aspect of GQM. The method provides little guidance, relying instead upon the judgment, experience and insight of those

involved with measurement to identify useful questions. There exists a multiplicity of questions that could be asked about virtually any goal. The problem is choosing those questions likely to shed light, or to support achievement of that goal. The top-down nature of GQM can also lead to difficulties. A goal may be so ambitious — for instance to eliminate all software defects prior to release - that it cannot readily be addressed by a manageable number of questions. Another problem is fuzzy goals. Despite the template it is difficult to completely eradicate goals such as "improve software quality from the perspective of the development organization". In such circumstances it would be better to backtrack and refine the goal than to continue to try to identify measurement questions.

Once the questions have been chosen, they must be quantified, so the question Q1 requires metrics to accomplish the following:

1. Measure specification "size" (e.g. function points).
2. Measure project costs (e.g. person hours and staff grade).

Although, a less difficult step than devising questions, identifying metrics is not a mechanistic step. In the above example, one could list a considerable number of candidate metrics for specification "size". Possibilities are function points, size of the specification in pages or words, the number of symbols if using a diagrammatic notation, DeMarco's "bang" metric and so forth. Choice should be governed by a combination of "best practice" and what is feasible on pragmatic grounds. The "bang" metric might be highly regarded in the literature, but if neither data flow nor entity-relationship diagrams are used as development aids, then collection of this metric could prove to be prohibitively expensive.

In order to give an example of application of the Goal/Question/Metric approach; let's suppose we want to improve the timeliness of change request processing during the maintenance phase of the life cycle of a system. The resulting goal will specify a purpose (improve), a process (change request processing), a viewpoint (project manager), and a quality issue (timeliness). This goal can be refined to a series of questions, about, for instance, turn-around time and resources used. These questions can be answered by metrics comparing specific turn-around times with the average ones. The complete Goal/Question/Metric Model is shown in Table 1.2.

Table 1.2: The GQM Approach

Goal	Purpose, Issue, Object (process), Viewpoint	Improve the timeliness of change request processing from the project manager's viewpoint.
Question		What is the current change request processing speed?
Metrics		Average cycle time, standard deviation % cases outside of the upper limit.
Question		Is the performance of the process improving?
Metrics		$\frac{\text{Current average cycle time}}{\text{Baseline average cycle time}} \times 100$ Subjective rating of manager's satisfaction

Check Your Progress

1. What are the factors for software crisis?
2. Explain the evolution of software role?

1.5 LETS SUM UP

Software engineering is an engineering discipline, which is concerned with all aspects of software production. Software products consist of developed programs and associated documentation. Essential product attributes are maintainability, dependability, efficiency and usability. The role of software has undergone drastic change in the last few decades. These improvements range through hardware, computing architecture, memory, storage capacity and a wide range of unusual input and output conditions. Size, cost, quality and delayed delivery are some factors of software crisis. Software have many advantages also like they are easy to change, more reliable, testing software removes all the errors etc. Software metrics measures the software product and the process from which it has been developed.

1.6 KEYWORDS

OS: Operating System

GUI: Graphical User Interface

SEI: Software Engineering Institute

Software: Software is a common term for the various kinds of programs used to operate computers and related devices.

Software Engineering: Software engineering deals with building and maintaining software systems.

Software Crisis: This term is used to describe the impact of rapid increases in computer power and the complexity of the problems.

Software Metrics: It is the measurement of software product and the process from which it has been developed.

GQM: Goal Question Metric

QFD: Quality Function Deployment

LOC: Line of Code

1.7 QUESTIONS FOR DISCUSSION

1. What do mean by software engineering?
2. Discuss various myths of software.
3. Define software metrics. In how many ways software metrics can be classified?
4. Explain GQM Approach.

Check Your Progress: Model Answers

1. Some common basic myths for software are easy to change, reliable, testing software are there to remove errors.
2. Software metrics is the process of measuring software product and the process from which it is developed.

1.8 SUGGESTED READINDS

R.S. Pressman, *Software Engineering-A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Rajib Mall, *Fundamentals of Software Engineering*, PHI, 2nd Edition.

Sommerville, *Software Engineering*, Pearson Education, 6th Edition.

Richard Fairpy, *Software Engineering Concepts*, Tata McGraw Hill, 1997.

LESSON

2

SOFTWARE LIFE CYCLE MODELS

CONTENTS

2.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Discuss software development life cycle models
- Identify Capability Maturity Model (CMM)
- Explain ISO 9000, ISO 9001, ISO 9002
- Understand common features and differences between CMM and ISO

2.1 INTRODUCTION

Building software is a continuous learning process and the outcome is nothing but a refined version of knowledge that has been collected and processed in the process. A software lifecycle model can be called a framework of tasks required to develop and build high-quality software. Can we call the software lifecycle model as software engineering? The answer is "yes" and "no". Software lifecycle model is techniques that are involved while software is being engineered. Also, the technical methods and tools that comprise in the software engineering form a part of the model. Software must be developed keeping in mind the demands of the end-user using defined software. We will discuss the various stages that go into the making of software and the software models involved therein, in this lesson. A structured set of activities required to develop a software lifecycle model:

- Specification
- Design
- Validation/Verification
- Evolution

A software lifecycle model is an abstract representation of a process. It presents a description of a process from some particular perspective.

2.2 SOFTWARE LIFE CYCLE MODELS

A software life cycle model or process model is a simplified representation of a software process, presented from a specific perspective. Various examples of process perspectives are:

- Workflow perspective - sequence of activities
- Data-flow perspective - information flow
- Role/action perspective - who does what

The linear sequential model or the waterfall model is based on the assumption that a complete system is delivered at the end of the complete software development cycle. It is designed for linear development. The prototype model is designed to make the developer and the customer understand the requirements in a better way. It does not deliver a production system at once. However, these models are not evolutionary i.e. iterative in nature. Iterative models help the software engineers to develop more complex software.

2.2.1 Build and Fix Model

Build and Fix model is the most basic and simple model of software development which does not have any specifications, nor any effort to design and generally testing is also neglected. This is a representation of what is happening in many software development projects.

- Not an appropriate model
- Implementation of system without specification, design and testing.
- May work for small scale projects

- Code is written, then modified until client is happy
- Very Risky

This model, shown in Figure 2.1, is not the perfect model to develop a software project. The model is constructed without any appropriate specifications and design steps. In fact, the product is built and modified as many times as probable until it satisfies the client. The cost of build-and fix is actually far greater than the cost of properly specified and carefully designed product. Software engineers are strongly discouraged from using this development.

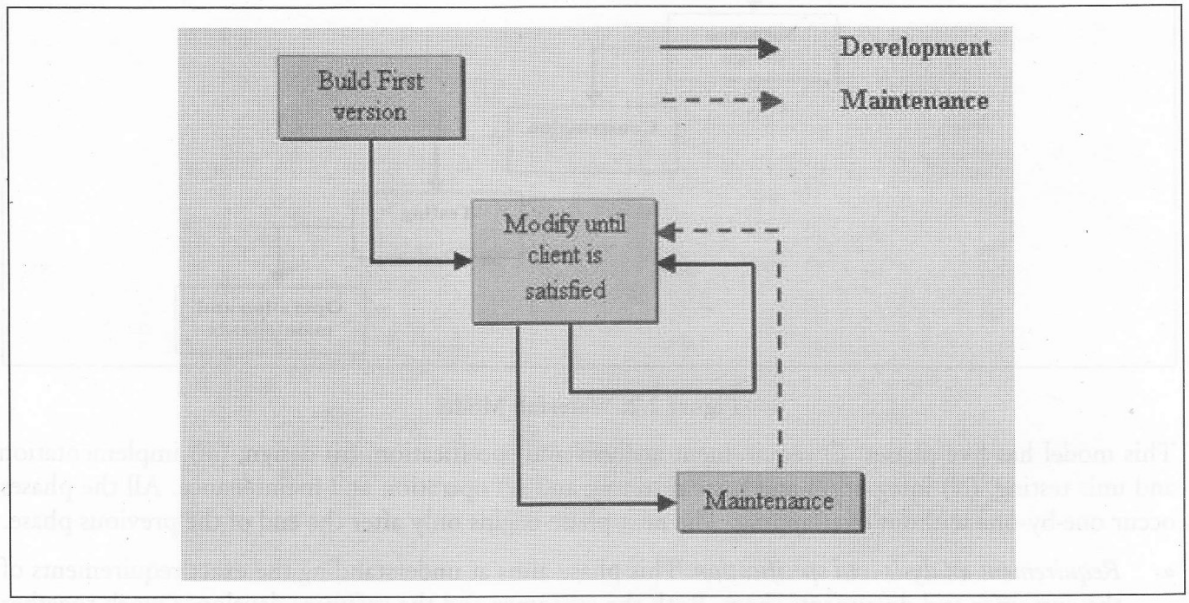

Figure 2.1: Build and Fix Model

It is unfortunate that many products are developed using the build-and-fix model. The developers simply build a product that is reworked as many times as essential to satisfy the client. This model may work for small projects but is totally unacceptable for products of any reasonable size.

Advantages

- Cost efficient for very small projects of limited complexity.

Disadvantages

- Indecisive approach for products of sensible size.
- Cost is higher for bigger projects.
- Mostly product will not be delivered on time.
- Often results in a product are generally of low quality.
- No documentation is created.
- Maintenance can be very difficult without specification and design document.

2.2.2 Waterfall Model

The waterfall model derives its name due to the cascading effect from one phase to the other as is illustrated in the Figure 2.2. In this model each phase is well defined, has a starting and ending point, with identifiable deliveries to the next phase.

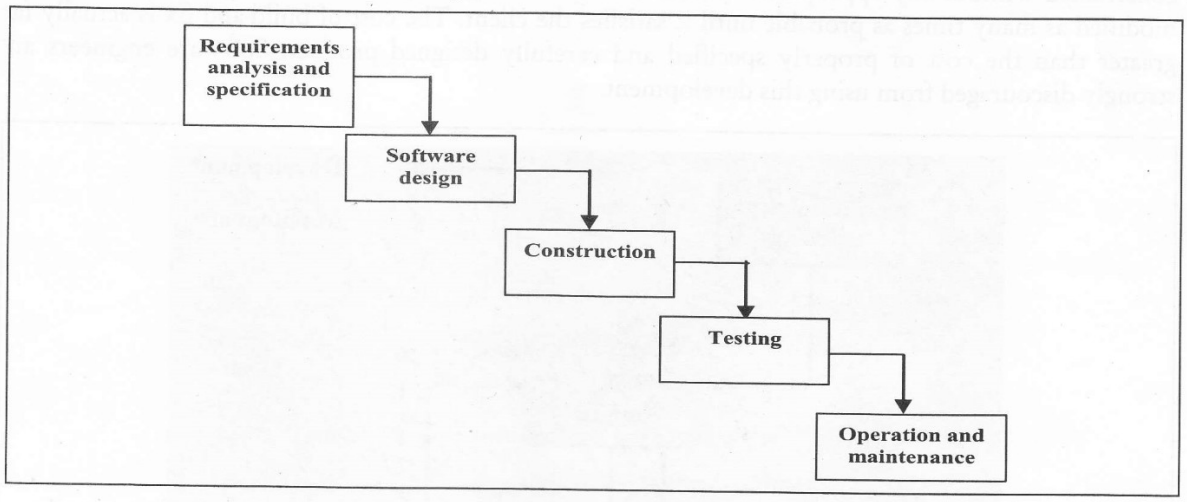

Figure 2.2: Waterfall Model

This model has five phases: (i) requirement analysis and specification, (ii) design, (iii) implementation and unit testing, (iv) integration and system testing and (v) operation and maintenance. All the phases occur one-by-one without overlapping. The next phase begins only after the end of the previous phase.

- *Requirement analysis and specification:* This phase aims at understanding the exact requirements of the customer and documents them. Both the customer and the software developer work together so as to document all the functions, performance and interfacing requirements of the software. It describes the "what" of the system to be produced and not "how". In this phase a large document called Software Requirement Specification document (SRS) is produced which contains the detailed description of what the system will do in the common language.
- *Design phase:* The aim of this phase is to transform the requirements gathered in the SRS into a suitable form which permits further coding in a programming language. It defines the overall software architecture together with high level and detailed design. All this work is documented as Software Design Description document (SDD).
- *Implementation and unit testing phase:* The actual coding begins at his stage. The implementation goes smoothly if the SDD has complete information required by the software engineers.

  During testing, the code is thoroughly examined and modified. Small modules are testes in isolation initially. Thereafter, these modules are tested by writing some overhead code in order to check the interaction between these modules and the flow of intermediate output.
- *Integration and system testing:* This phase is highly crucial as the quality of the end product is determined by the effectiveness of the testing carried out. Better output will lead to satisfied customers, lower maintenance costs and accurate results. Unit testing determines the efficiency of individual modules. However, in this phase the modules are tested for their interactions with each other and with the system.

- *Operation and maintenance phase:* Maintenance is the task performed by every user once the software has been delivered to the customer, installed and operational.

Thus, the delivery of software initiates the maintenance phase. The time and efforts spent on the software to keep it operational after release is important. It includes activities like error correction, removal of obsolete functions, optimization and enhancements of functions. It may span for 5 to 50 years.

Thus, in this model the requirements must be clear at the very initial stages. The end product is available relatively late which in turn delays the error recovery.

Project Outputs in Waterfall Model

As we have seen, the output of a project employing the waterfall model is not just the final program along and documentation to use it. There are a number of intermediate outputs that must be produced to produce a successful product. Though the set of documents that should be produced in a project is dependent on how the process is implemented, the following is a set of documents that generally forms the minimum set that should be produced in each project:

- Requirements document
- Project plan
- System design document
- Detailed design document
- Test plan and test reports
- Final code
- Software manuals (e.g., user, installation, etc.)
- Review reports

Except for the last one, these are the outputs of the phases, and they have been briefly discussed. To certify an output product of a phase before the next phase begins, reviews are often held. Reviews are necessary, especially for the requirements and design phases, because other certification means are frequently not available. Reviews are formal meetings to uncover deficiencies in a product and will be discussed in more detail later. The review reports are the outcome of these reviews.

Limitations of the Waterfall Model

Software life-cycle models of the waterfall variety are among the first important attempts to structure the software life cycle. However, the waterfall model has limitations. Like a waterfall, progress flows in one direction only, towards completion of the project (from one phase to the next). Schedule compression relaxes this requirement but introduces new complexities.

Well-defined phases, frozen deliverables, and formal change control make the waterfall model a tightly controlled approach to software development. The waterfall model's success hinges, however, on the ability:

- To set the objectives.
- To state requirements explicitly.
- To gather all the knowledge necessary for planning the entire project in the beginning.

The waterfall model's original objectives were to make small, manageable, individual development steps (the phases) and to provide enough control to prevent runaway projects. There are a few problems that have caused dissatisfaction with this phased life-cycle approach.

- The new software system becomes useful only when it is totally finished, which may create problems for cash flow and conflict with organizational (financial) objectives or constraints. Too much money may be tied up during the time the software is developed.
- Neither user nor management can see how good or bad the system is until it comes in. The users may not have a chance to get used to the system gradually.
- Changes that "aren't supposed to happen", are not viewed kindly, neither for requirements nor during the operational life of the system. This can shorten the software's useful life.

Because of these shortcomings, other process models have appeared. They are based on the concept of iteration and evolution. Despite these limitations, the waterfall model is the most widely used process model. It is well suited for routine types of projects where the requirements are well understood. That is, if the developing organization is quite familiar with the problem domain and the requirements for the software are quite clear, the waterfall model works well.

2.2.3 Prototype Model

A prototype model is beneficial when the customer requirements are dynamic and keep on changing with time and the developer is unsure about the software adaptability with the system and the operating system. Thus in a prototype model, a working prototype is built with the available set of requirements such that it has limited functionalities, low reliability and performance.

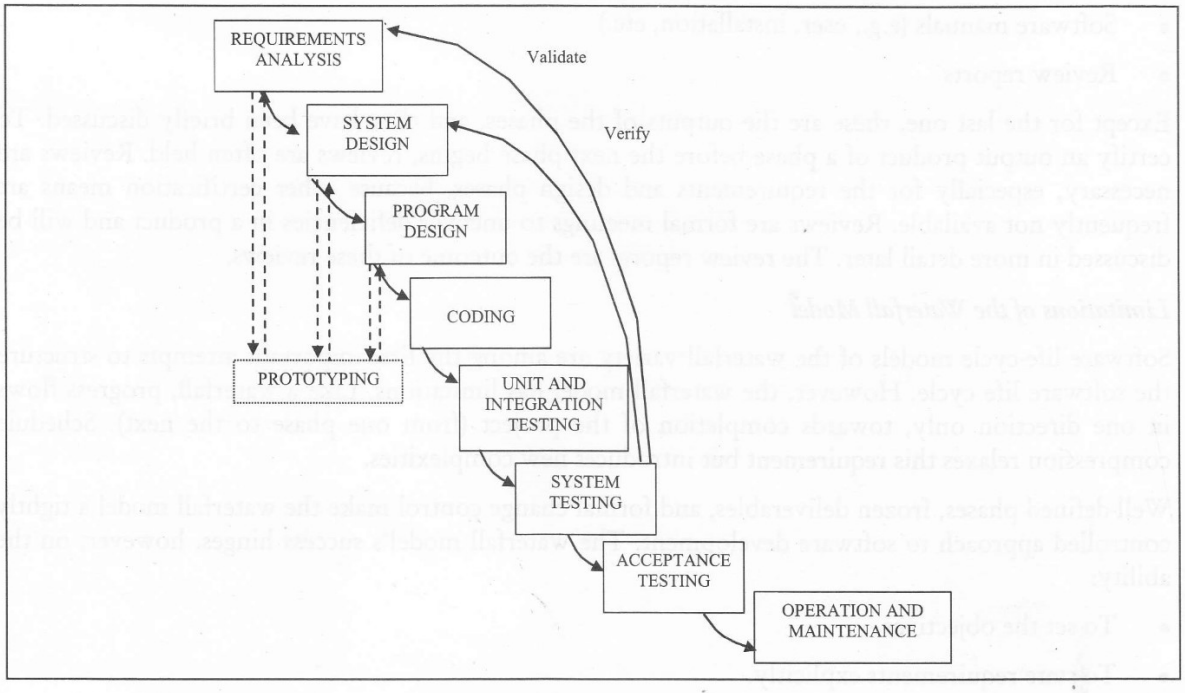

Figure 2.3: Prototype Model