NEERAJ®

# SOFTWARE ENGINEERING

*By:*

**Sumeet Sharma**

B.E. (Computer Science)

*New Edition*

**23260329**
**65817881**
**Fax : 011-23285501**

**NEERAJ PUBLICATIONS**

***(Publishers of Educational Books)***

***1507, Ist FLOOR, NAI SARAK, DELHI - 110 006***

*E-mail:* **info@neerajbooks.com**

*Website:* **www.neerajbooks.com**

**Price: Rs. 160.00**

**_Published by:_**

**NEERAJ PUBLICATIONS**

*Admn. Office* : **Delhi-110 007**

*Sales Office* : **1507, 1st Floor, Nai Sarak, Delhi-110 006**

*E-mail* : **info@neerajbooks.com** *Website* : **www.neerajbooks.com**

*Typesetting by* : ***Competent Computers*** *Printed at :*

# CONTENTS

## SOFTWARE ENGINEERING

❐❐

# SOFTWARE ENGINEERING

AN OVERVIEW OF SOFTWARE ENGINEERING

# Software Engineering and its Models 1

The evolution of electronic computers began in 1940s. Early efforts in the field of computing were focused on designing the hardware, as that was the challenge, and hardware was where most technical difficulties existed. In the early computing systems, there is essentially no operating system and the programs were fed with paper tapes or by switches. There was a gradual trend towards isolating the user from the machine internals, so the user could concentrate on solving the problem at hand, rather than getting bogged down with the machine details.

In the period, when higher programming languages like PASCAL, COBOL came into existence, the use of these made programming easier. Some of the structural design practices like top-down approach were introduced. The concept of quality assurance was also introduced. However, the business aspects like cost estimation, time estimation, etc. of the software were in their elementary stages.

Later on programming team in an organization, full-fledged software companies evolved. A software houses primary business is to produce software. As software houses offered a range of services, including hiring out of suitably qualified personnel to work within client's team, consultancy and a complete system design and development service. The output of these companies was *Software*. They viewed the software as a product and its functionality as the process. With the coming of the multiprogramming operating systems, the usability and efficiency of the computing machines took a big leap. This is when software became more strategic, disciplined and commercial.

Now with the advancement of technology, the cost of hardware is consistently decreasing. On the other hand, the cost of software is increasing. The main reason for the high cost of software is that software technology is still labour-intensive. Software projects are often very large, involving many people, and span over many years. The development of these systems is often done in an *ad-hoc* manner, resulting in frequent schedule slippage and cost over runs in software projects. Some projects are simply abandoned. Customized software fails because the design fails. If a design fault is detected in software, changes are usually made to remove that fault so that it causes no failures in future. Here comes the need of Software Engineering.

Let us first discuss what is meant by software in context of software engineering. Software is not merely a collection of computer programs. There is a distinction between a program and a programming system product. A program is generally complete in itself, and is generally used only by the author of the program. There is usually little documentation or other aids which can help other people use the program. Since the author is the user, the presence of bugs is not a major concern; if the program crashes, the author will fix the program and start using it again. These programs are not designed with such big issues as portability, reliability and usability in mind.

However, a programming system product is used largely by people other than the developers of the system. The users may be from different backgrounds, so a proper user interface is provided. There is sufficient documentation to help these diverse users use the system. Programs are thoroughly tested before put into operation use, because users do not have the luxury of fixing bugs that may be detected. Since the product may be used in a variety of environments, perhaps on a variety of hardware, portability is a key issue.

***Software*** is a collection of computer programs, procedures, rules and associated documentation and data.

***Software Engineering*** is the systematic approach to the development, operating, maintenance and retirement of the software.

***Or***

***Software Engineering*** is the application of science and mathematics by which the capabilities of computer equipments are made useful to the man *via* computer programs, procedures, and associated documentation.

The basic goal of software engineering is to produce high quality software at low cost. The two basic driving factors are quality and cost. Cost of a completed project can be calculated easily if proper accounting procedures are followed. Quality of the software is something not so easy to qualify and measure.

We can specify three dimensions of the product whose quality is to be assessed:

**1. Product Operations:** The first factor of product operation deals with quality factors such as correctness, reliability, and efficiency.

*(i)* *Correctness.* It is the extent to which a program satisfies its specifications.

*(ii)* *Reliability.* It is the property which defines how well the software meets its requirements.

*(iii)* *Efficiency.* It is a factor in all issues relating to the execution of software and includes such considerations as response time, memory requirements and throughput.

*(iv)* *Usability.* It is the effort required to learn and operate the software properly. Hence it is an important property that emphasizes the human aspect of the system.

**2. Product Transition:** The second factor of product transition deals with equality factors like portability and interoperability.

*(i)* *Portability*. It is the effort required to transfer the software from one hardware configuration to another.

*(ii)* *Reusability.* It is the extent to which parts of the software can be reused in other related applications.

*(iii)* *Interoperability.* It is the effort required to couple the system with other systems.

**3. Product Revision:** The product revision is concerned with those aspects related to modification of programs and includes factors like maintainability, flexibility and testability.

*(i)* *Maintainability*. It is the effort required to locate and fix errors in operating programs

*(ii)* *Flexibility*. It is the effort required to modify an operation program (perhaps to enhance its functionality).

*(iii)* *Testability*. It is the effort required to test, to ensure that the system or a module performs its intended function.

## SOFTWARE PROCESS TECHNOLOGY

When we develop a program or build something, there are some activities we perform, either explicitly or implicitly. Suppose that Mr. X, a computer programmer, buys a car, and wants to write a program to verify that his car payments are correct. With this eventual goal, what is the first thing X will do? The first natural thing to do is to try to understand the problem better and more precisely. He will identify that the inputs are the cost of the car, the interest rate, the duration of the loan, and the monthly payment the car dealer has told him. The goal is to determine, given these inputs, if the payment is consistent with the other inputs. Mr. X may do this problem definition implicitly in his mind and without realizing it. However, the problem must be clearly understood by the programmer before he starts coding.

The next step that X will take is to decide what course he should follow—should he determine the payments and then compare, or should he use the payments as data and determine the rate and check the rate? What algorithm should he use? In other words, before he starts coding, he has to plan in his mind about how to solve the problem. What will he code, if he does not know what he is trying to code? Again this activity of deciding a plan for a solution may be done implicitly by Mr. X perhaps while sitting on the terminal and in parallel with coding itself. However, a plan for a solution must be thought of. Once he has coded the plan, he will try to test and debug it.

From this situation we can say that problem solving in software must consists of these activities—

understanding the problem, deciding a plan for a solution, coding the planned solution, and finally testing the actual program. For small problems, these activities may not be done explicitly. The start and end boundaries of these activities may not be clearly defined and no written record of the activities may be kept. However, for large systems where the problem solving activity may last over a few years, and where many people are involved in development, performing these activities implicitly without proper documentation and representation will clearly not work.

For any software system of a non-trivial nature, each of the four activities for problem solving listed above has to be done formally. For large systems, each activity can be extremely complex and methodologies and procedures are needed to perform it efficiently and correctly. Each of these activities is a major task for large software projects. Furthermore, each of the basic activities itself may be so large that it cannot be handled in a single step and must be broken into smaller steps. For example, design of a large software system is almost always broken into multiple distinct design phases, starting from a very high level design specifying only the components in the system to a detailed design where the logic of the components is specified. The basic activities or phases to be performed for developing a software system are:

**REQUIREMENT ANALYSIS**

This phase is essential in order to understand the problem which the software system is to solve. The problem could be automating an existing manual process, or developing a completely new automated system, or a combination of the two. The emphasis is on identifying what is needed from the system and not that how the system will achieve its goals. The goal of this phase is to produce the software requirement specification document. The requirements document must specify all functional and performance requirements, the doormats of inputs, outputs and any required standards, and all design constraints that exist due to political, economic, environmental, and security reasons.

**SOFTWARE DESIGN**

The purpose of the design phase is to plan a solution of the problem specified by the requirements document. In other words, starting with *what is needed*, design takes us towards *how to satisfy the needs*? The design of a system is perhaps the most critical factor affecting the quality of the software and has a major impact on the later phases. The output of this phase is the design document. The design activity is often divided into two separate phases:

**1. System design:** It is sometimes called as top-level design, as it aims to identify the modules that should be in the system, the specifications of these modules, and how they interact with each other to produce the desired results. A *design methodology* is a systematic approach to creating a design by application of a set of techniques and guidelines. Most methodologies focus on system design. The two basic principles used in any design methodology are the problem partitioning and abstraction.

***(i)*** **Partitioning:** A large system cannot be handled as a whole, and so for design it is partitioned into smaller systems. This partitioning process can continue further till we reach a stage where the components are small enough to be designed separately. This divide and conquer method is essential for handling large projects and all design methodologies provide methods to partition the problem effectively.

***(ii)*** **Abstraction:** Abstraction is a concept related to problem partitioning. When is used during design, the design activity focuses on one part of the system at a time. Since the part being designed interacts with other parts of the system, a clear understanding of the interaction is essential for properly designing the part. An abstraction of a system or a part defines the overall behaviour of the system at an abstract level without giving the internal details. While working with a part of the system, a designer needs to understand only the abstractions of the other parts with which the part being designed interacts. The use of abstraction allows the designer to practice the divide and conquer technique effectively by focusing on one part at a time, without worrying about the details of other parts.

**2. Detailed design:** The internal logic of each of the modules specified in system design is decided. During this phase further details of the data structures and algorithmic design of each of the modules is specified. The logic of a module is usually specified in a high-level design description language, which is independent of a target language in which the software will eventually be implemented.

The design phase ends with verification of the design. If the design is not specified in some executable languages, the verification has to be done by evaluating the design documents. One way of doing this is through reviews.

## CODING

The goal of the coding phase is to translate the design of the system into code in a given programming language. For a given design, the aim in this phase is to implement the design in the best possible manner. The coding phase affects both testing and maintenance profoundly. A well written code can reduce the testing and maintenance effort. Since the testing and maintenance costs of software are much higher than the coding cost, the goal of coding should be to reduce the testing and maintenance effort. An important concept that helps the understandability of programs is structured programming. The goal of structured programming is to linearize the control flow in the program. The program text should be organized as a sequence of statements and during execution the statements are executed in the sequence given in the program. For structured programming, a few single-entry-single-exit constructs should be used. These constructs include selection (if-then-else), iteration (while-do, repeat-until, etc.). With these constructs it is possible to construct a program as a sequence of single-entry-single-exit constructs.

In coding phase, the entire system is not tested together. Rather the different modules are tested separately. This testing of modules is called as "Unit-testing". The output of these phase is the verified and unit tested code of different modules.

## TESTING

Testing is the major quality control measure employed during software development. Its basic function is to detect errors in the software. This phase has to uncover errors during the coding, but also errors introduced during the previous phases. The goal of testing is to uncover requirement, design and coding errors in the programs. The starting point is off-course the **unit testing**, but gradually the modules are integrated into sub-systems and are then subjected to the **integration testing**. After the system is ready, the **system testing** is performed. The whole of the system is tested against the system requirements. Finally, the **acceptance testing** is performed to demonstrate to the clients, on the real life data of the client. For testing to be successful, proper selection of test cases is essential. There are two different approaches to selecting test cases—functional testing and structural testing. In *functional testing* the software or the module to be tested is treated as a black box and the test cases are decided based on the specifications of the system or the module. In structural testing the test cases are decided based on the logic of the module to be tested. A common approach here is to achieve some type of coverage of the statements in the code. In *structural testing* the test cases are decided based on the logic of the module to be tested. A common approach here is to achieve some type of coverage of the statements in the code. One common coverage criterion is statement coverage which requires that test cases be selected so that together they execute each statement at least once.

The following are some guidelines for testing:

*(i)* Test the modules thoroughly, cover all the access paths, generate enough data to cover all the access paths arising from conditions.

*(ii)* Test the modules by deliberately passing wrong data.

*(iii)* Specifically create data for conditional statements. Enter data in test file which would satisfy the condition and again test the script.

*(iv)* Test for locking by invoking multiple concurrent processes.

The following objectives are to be kept in mind while performing testing:

*(i)* It should be done with the intention of finding errors.

*(ii)* Good test cases should be designed which have a portability of finding, as yet undiscovered error.

*(iii)* A success test is one that uncovers yet undiscovered error(s).

The following are some of the principles of testing:

*(i)* All tests should be performed according to user requirements.

*(ii)* Planning of tests should be done long before testing.

*(iii)* Starting with a small test, it should proceed towards large tests.