Software Engineering and Project Management

INTRODUCTION

Computers have been used for commercial purposes for the fifty years. Initially, computers were very slow and lacked sophistication. The computational power and sophistication of computers have increased ever since, while their prices have reduced dramatically. The improvements in their speed and reductions in their costs have been the result of several technological breakthroughs which occurred at regular intervals.

The more powerful a computer is the more sophisticated programs it can run. Therefore, with every aspect of improvement in the of computers, software engineers have been tasked to solve larger and more complex problems and that too in cost effective and efficient way. Software engineers have admirably coped with this challenge by innovating and by building upon their past programming experience. All those past innovations and experiences of writing good quality programs in cost-effectives and efficient ways have been systematically organized into a body of knowledge. This knowledge forms the foundation of the software engineering principles. From this point of view, we can say that the discipline of software engineering discusses **systematic and cost-effective software development approaches** which have come about from past innovations and lessons learnt from mistakes.

Alternatively, we can view software engineering as the engineering approach to develop software. In earlier times, software was simple in nature and hence, software development was a simple activity. However, as technology improved, software became more complex and software projects grew larger. Software development now necessitated the presence of a team, which could prepare detailed plans and designs, carry out testing, develop intuitive user interfaces, and integrate all these activities into a system. This new approach led to the emergence of a discipline known as **software engineering.** What exactly is an **engineering approach** to develop software? Let us try to answer this question using an analogy. Suppose you ask a petty contractor to build a small house for you. Petty contractors are not really expert's in house building. They normally carry out minor repair works and at most undertake very small building works such as the construction of boundary walls, etc. Now faced with the task of building a complete house, your petty contractor would draw upon all the limited knowledge he has of house building. Yet, he would often be left with no clue as to what to do. For example, he might not know the optimal proportion of cement and sand mix to realize sufficient structural strength. In such situations, he would have to fall back upon his intuitions. He would most probably succeed in his work, if the house you asked him to construct was sufficiently small. Even this small house constructed by him would perhaps not look as good as the one constructed by a professional. The house might lack proper planning, display several defects and imperfections. It may even cost more and take longer to build.

Now, suppose you entrust your petty contractor to build a large 50-storeyed commercial complex for you. He might exercise prudence, and politely refuse to undertake your request. On the other hand, he might be ambitious and agree to undertake the task. In the later case, he is sure to fail. The failure might come in several forms: the building might collapse during the construction stage itself due to his ignorance of the theories concerning the strengths of materials; the construction might get unduly delayed, If he does not prepare proper estimations and delayed plans regarding the type and quantity of raw materials required, the times at which these may be required, and so forth. In short, to be successful in constructing a building of large magnitude, one needs a thorough knowledge of civil and architectural engineering techniques such as analysis, estimation, prototyping, planning, designing, testing and so on. Similar things happen in case of software development projects. For sufficiently small sized problems, one might processed according one's intuition and succeeds. Of course ,the solution might have several imperfections ,cost more ,take longer to complete ,and so forth .But failure is almost certain if one undertakes a large scale-software development work without sound applications of the software engineering principles.

From the building construction analogy, we can say that there exit several fundamental similarities between software engineering and other engineering approaches such as civil engineering. These are characterized by the following aspects:

- Heavy use of past experiences. The past experiences are systematically arranged and theoretical basis for them are also provided wherever possible. Whenever no reasonable theoretical justification can be provided, the past experiences are adopted as rules of thumb.
- While designing a system, several conflicting goals might have to be optimized. In such situations, there may not be any unique solution and thus several alternative solutions may be proposed. While selecting an appropriate solution, various trade-offs are considered. To arrive at the final design, several iterations and backtracking may have to be performed.
- A pragmatic approach to cost-effectiveness is adopted and economic concerns are addressed.

SOFTWARE

Software is defined as a collection of programs, documentation and operating procedures. The Institute of Electrical and Electronic Engineers (IEEE) defines software as a 'collection of computer programs, procedures, rules and associated documentation and data'. It is responsible for controlling, integrating and managing the hardware components of a computer and to accomplish specific tasks. In other words, software tells the computer **what to do** and **how to do** it. For example, software instructs the hardware what to display on the user's screen, what kinds of input to take from the user, and what kinds of output to generate. Thus, software communicates with the hardware by organizing the control sequences, and the hardware carries out the instructions defined by the software.

A set of programs, which are specifically written to provide the user a precise functionality likes solving a specific problem is termed as a software package. For example, word processing software package provides functionality to the computer so that it can be used to create text documents like letters and mailing lists. Similarly, an image processing software package assists a user in drawing and manipulating graphics.

SOFTWARE CHARACTERISTICS

Different individuals judge software on different basis. This is because they are involved with the software in different ways. For example, users want the software to perform according to their requirements. Similarly, developers involved in designing, coding and maintenance of the software evaluate the software by looking at its internal characteristics, before delivering it to the user. Software characteristics are classified into six major components, which are shown in figure.

1. **Functionality:** Refers to the degree of performance of the software against its intended purpose.
2. **Reliability:** Refers to the ability of the software to provide desired functionality under the given conditions.
3. **Usability:** Refers to the extent to which the software can be used with ease.
4. **Efficiency:** Refers to the ability of the software to use system resources in the most effective and efficient manner.
5. **Maintainability:** Refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors.
6. **Portability:** Refers to the ease with which software developers can transfer software from one platform to another, without (or with minimum) changes. In simple terms, it refers to the ability of software to function properly on different hardware and software platforms making any changes in it.

In addition to the above mentioned characteristics, robustness and integrity are also important. Robustness refers to the degree to which the software can keep on functioning in spite of being provided with invalid data while integrity refers to the degree to which unauthorized access to the software or data can be prevented.

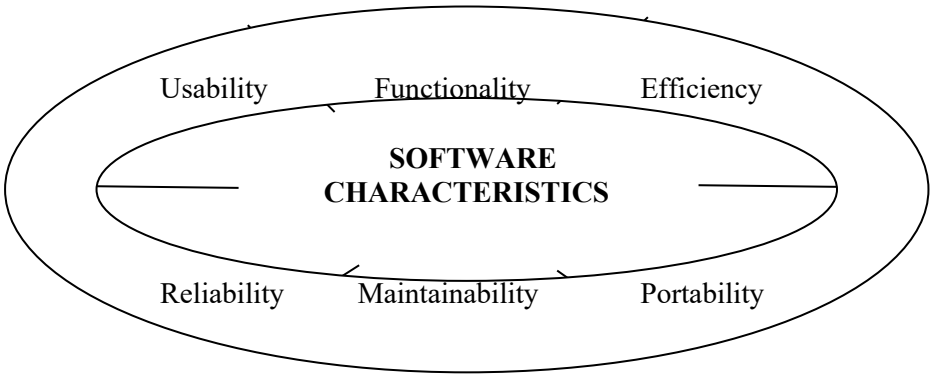

Figure: Software Characteristics

CLASSIFICATION OF SOFTWARE

Software can be applied in countless fields such as business, education, social sector and other fields. It is designed to suit some specific goals such as data processing, information sharing, communication, and so on. It is classified according to the range of potential of applications. These classifications are listed below.

- **System software:** This class of software manages and controls the internal operations of a computer system. It is a group of programs, which is responsible for using computer resources efficiently and effectively. For example, an operating system is system software, which, controls the hardware, manages memory and multitasking functions, and acts as an interface between application programs and the computer.
- **Real-time software:** This class of software observes, analyze, and controls real world events as they occur. Generally, a real time system guarantees a response to an external event within a specified period of time. An example of real-time software is the software used for weather forecasting that collects and processes parameters like temperature and humidity from the external environment to forecast the weather. Most of the defence organizations all over the world use real-time software to control their military hardware.
- **Business software:** This class of software is widely used in areas where management and control of financial activities is of utmost importance. The fundamental component of a business comprises payroll, inventory, and according software that permit the user to access relevant data from the database. These activities are usually performed with the help of specialized business software that facilities efficient framework in business operations and in management decisions.
- **Engineering and scientific software:** This class of software has emerged as a powerful tool in the research and development of next generation technology. Applications such as the study of celestial bodies, under-surface activities, and programming of an orbital path for space shuttles are heavily dependent an engineering scientific software. This software is designed to perform precise calculations on complex numerical data that are obtained during real-time environment.
- **Artificial intelligence (AI) software:** This class of software is used where the problem solving technique is non algorithmic in nature. The solutions of such problems are generally non-agreeable to computation or straightforward analysis. Instead, these problems require specific problem-solving strategies that include expert system, pattern recognition, and game playing techniques. In addition, they involve different kinds of search techniques which include the use of heuristics. The role of artificial intelligence software is to add certain degrees of intelligence to the mechanical hardware in order to get the desired work done in an agile manner.

- **Web-based software**: This class of software acts as an interface between the user and the internet. Data on the internet is in the form of text, audio, or video format, linked with hyperlinks. Web browser is a software that retrieves web pages from the internet. The software incorporates executable instructions written in special scripting languages such as CGI or ASP. Apart from providing navigation on the web, this software also supports additional features that are useful while surfing the internet.
- **Personal computer software:** This class of software used for both official and personal use. The personal computer software market has grown over in the last two decades from normal text editor to word processor and from simple paintbrush to advanced images-editing software. This software is used predominantly in almost every field, whether it is database management system, financial accounting package, or multimedia-based software. It has emerged as a versatile tool for routine applications.

Software Myths

There are number associated with software development community. Some of them really affect the way, in which software development should take place. In this section, we list few myths, and discuss their applicability to standard software development.

1. **Software is easy to change:** It is true source code files are easy to edit, but that is quite different than saying that software is easy to change. This is deceptive because source code is so easy to alter. But making changes without introducing errors is extremely difficult, particularly in organizations with poor process maturity. Every change requires that the complete system be re-verified. If we do not take proper care, this will be an extremely tedious and expensive process.
2. **Computers provide greater reliability than the devices they replace:** It is true that software does not fail in the traditional sense. There are no limits to how many times a given piece of code can be executed before it "wears out". In any event, the simple expression of this myth is that our general ledgers are still not perfectly accurate, even though they have been computerized. Back in the days of manual accounting systems, human error was a fact of life. Now, we have software error as well.
3. **Testing software or 'proving' software correct can remove all the errors:** Testing can only show the presence of errors. It cannot show the absence of errors. Our aim is to design effective test cases in order to find maximum possible errors. The more we test, the more we are confident about our design.
4. **Reusing software increases safety:** This myth is particularly troubling because of the false sense of security that codes re-use can create. Code re-use is a very powerful tool that can yield dramatic improvement in development efficiency, but it still requires analysis to determine its suitability and testing to determine if it works.
5. **Software can work right the first time:** If we go to an aeronautical engineer, and ask him to build a jet fighter craft, he will quote us a price. If we demand that it is to be put in production without building a prototype, he will laugh and may refuse the job. Yet, software engineers are often asked to do precisely this sort of work, and often accept the job.
6. **Software can be designed thoroughly enough to avoid most integration problems:** There is an old saying among software designers: "Too bad, there is no compiler for specifications". These points out the fundamental difficulty with detailed specifications. They always have inconsistencies, and there is no computer tool to perform consistency checks on these. Therefore, special care is required to understand the specifications, and if there is an ambiguity, that should be resolved before proceeding for design.
7. **Software with more features is better software:** This is, of course, almost the opposite of the truth. The best, most enduring programs are those which do one thing well.
8. **Addition of more software engineers will make up the delay:** This is not true in most of the cases. By the process of adding more software engineers during the project, we may further delay the project. This does not serve any purpose here, although this may be true for any civil engineering work.

9. **Aim is to develop working programs:** The aim has been shifted from developing working programs to good quality, maintainable programs. Maintaining software has become a very critical and crucial area for software engineering community.

This list is endless. These myths, poor quality of software, increasing cost and delay in the delivery of the software have been the driving forces behind the emergence of software engineering as a discipline. In addition, following are the contributing factors:

- Change in ratio of hardware to software costs
- Increasing importance of maintenance
- Advances in software techniques
- Increased demand for software
- Demand for larger and more complex software systems

Programs vs software products

Programs are developed by individuals for their personal use. They are, therefore, small in size and have limited functionality. Further, since the author of a program himself uses and maintains his programs, these usually lack good user-interface and proper documentation. For example, the programs developed by a student as part of his class assignments are programs and not software products. On the other hand, software products have multiple users and, therefore, have good user-interface, proper users, manuals, and good documentation support. Since, a software product has a large number of users; it is systematically designed, carefully implemented, and thoroughly tested. In addition, a software product consists not only of the program code but also of all associated documents such as the requirements specification document the design document, the test document, the users manuals, and so on. A further difference is that the software products often are too large to be developed by any individual. A software product is usually developed by a group of engineers working in a team.

We will distinguish between software engineers who develop software products and programmers who write programs. Since a group of software engineers usually work together in a team to develop a software product, it is necessary for them to adopt some systematic development methodology. Otherwise, they would find it very difficult to interface each other's work, produce a coherent set of documents, and understand each other's work.

Even though software engineering principles are indented primarily for use in the development of software products, many results of software engineering are appropriate for development of small programs as well. However, when developing small programs for personal use, rigid adherence to software engineering principles in often worthwhile. An ant can be killed using a gun, but it would be ridiculously inefficient and inappropriate. CAR Hoare [1994] has rightly observed that using software engineering principles to develop toy programs is very much like employing civil and architectural engineering principles to build sand castle for children to play.

Why study software engineering

Let us examine the skills you could be acquiring from a study of the software engineering principles. Possibly the most important skill you could be acquiring is the skill for developing large software products. We have seen that the exploratory program development style comes naturally to everybody, but is suitable for development of small programs only. The exploratory development approach breaks down when the size of the program increases. But, what are the reasons behind the exploratory style breaking down with increase in product size? A major part of the problem lies in the exponential growth in the perceived complexity and difficulty with program size if one attempts to write the program for a problem without suitable decomposing the problem. Consequently, the time and effort required to

develop a software product also grow exponentially if you are trying to develop it without decomposing it.

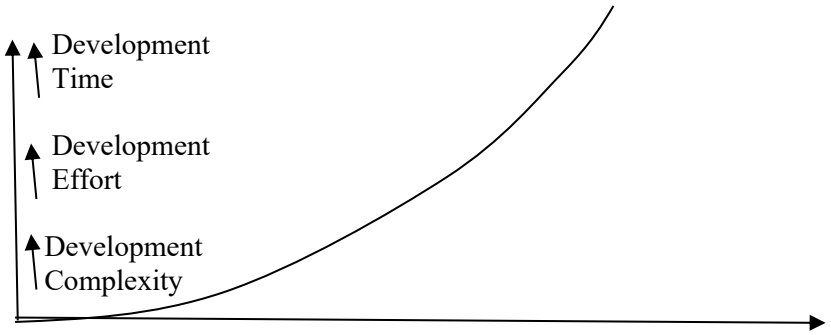

Figure: Increase in development time and effort with problem size

(But, why does the program development difficulty increase exponentially with the program size?) Unless we know how to decompose a problem into a set of smaller and independent parts, we would be overwhelmed by the problem. In fact, a major emphasis of every software design technique concerns how to effectively decompose a large problem into manageable parts. We would learn in this text that suitable problem decomposition to handle complexity is the essence of any good software design technique. Handling complexity in a software development problem is a central theme of the software engineering discipline. You would learn that abstraction and decomposition are two well accepted strategies that have been adopted in various ways to handle complexity. Besides these, you would also learn the techniques of software specification, user-interface development, testing, project management, and so forth. Even if you intend to write small programs, the software engineering principles could help you to achieve higher productivity and at the same time enable you to produce better quality programs.

Software Engineering

Software engineering is the discipline that covers principles of specification, systematic development, management and evolution of software system. It is concerned with the application of engineering concepts, techniques, and methods to the development of software.

Based on the preceding discussions, we can say that the basic objective of software engineering is to: *Develop methods and procedures for software development that can scale up large systems and that can be used to consistently produce high-quality software at low cost and with a small cycle time.* That is, the key objectives are consistency, low cost, high quality, small cycle time, and scalability.

The basic approach that software engineering tasks is to separate the development process from the developed product (i.e., the software). The premise is that the development process, or the software process, controls the quality, scalability, consistency, and productivity. Hence to satisfy the objectives, one must focus on the software process. Design of proper software processes and their control then becomes the primary goal of software engineering. It is this focus on process that distinguishes it from most other computing disciplines.

Most other computing disciplines focus on some type of product--algorithms, operating system, databases, etc.—while software engineering focuses on the process for producing the products. It is essentially the software equivalent of "manufacturing engineering."

There are several acceptable definitions of software engineering existing in the industry, some of which are explored as follows:

- **Definition 1:** *"Software Engineering is a systematisation of the process of software development to ensure the best solution most economically."*

This definition lays emphasis on two aspects of software engineering

- Systematic process of software development
- Economical solution that results from the use of methods, tools and procedures

- **Definition 2:** *"The application of a systematic disciplined, quantifiable approach to the development operation, maintenance of software: that is the application of engineering to software."* ***(IEEE)***

The above definition is more evolved. Apart from the systematic the economics factor, it speaks of the software development process becoming a disciplined (as opposed to chaotic) one as also one that is better controlled through measurement of various software parameters.

- **Definition 3:** *"Software Engineering is the application of science and mathematics by which the capabilities of a computer are made useful to man via computer programs, procedures and associated documentation."* ***(Bary Bohem)***

This definition looks at the application of science and mathematics to the process of software development.

- **Definition 4:** *"The systematic approach to the development, operation, maintenance and retirement of software."* ***(Charette-1986)***

This definition goes a step ahead and includes the retirement of software in its folds. Some modules of software become defunct and have to be retired; the others continue to function, needing a changed function every now and then. This means that software has to be continuously synchronized with the current business needs of a user.

- **Definition 5:** *The essence of software engineering is captured by Parnas (1989) who defines it as "multi-person construction of multi-version software."*

This definition highlights the essential difference between a programmer and a software engineer. While the former writes programs, the letter writes a software component that will be able to communicate with the components written by other software engineers to make a system.

The important thing to remember about this component is that it may be reused by others, modified by them, and plugged into different, through appropriate parts of a modified system and yet it must perform. These components are continuously revised and changed by multiple people thereby making the software that they are a part of multi-version as mentioned above.

It ensures that the method followed in developing software, is such that the end product is a quality. We build skyscrapers and are confident that they will not collapse. This is because these have been constructed using Civil Engineering principles. Civil Engineering outlines methods and procedures to be followed that ensures buildings are strong.

Software Engineering Management

Since the size and complexity involved in software has gone up drastically, need for more workforces have also increased. To manage the needs of all the issues confronting software and the individuals involved in it, software engineering is associated with the management in order to manage tasks such as preparing the development plan, project planning and cost estimation, and organizing people. The software cost is typically estimated in terms of total effort (person-month) required for developing the software.

Other management issues arise in developing a programming process that determines what is done, where, and by whom. This includes documentation required at each stage of development, coding conventions to be used, design and code review to be conducted and how testing is performed.

The Software Engineer

The software engineer is an individual who is responsible for performing all the software engineering activities including analysis, designing, coding, testing, and maintenance. In addition, he is also responsible for maintaining subsystems and external interface as well as integrating subsystems into a complex software system. To perform all these activities, a software engineer should possess the following skills.

- **Problem solving skills**: A software engineer should be able to develop algorithms and solve programming problems.
- **Programming skills:** A software engineer should have knowledge of data structures and algorithms, programming languages, and software tools. In addition, he should possess strong programming capabilities.
- **Design skills:** A software engineer should be familiar with numerous design approaches required during the development of software. He should be capable of extracting and specifying requirements precisely from ambiguous and unclear requirements. In addition, it is also expected from a software engineer to be able to switch between different levels of abstraction, from requirements specifications to design to coding.
- **Project management skills:** A software engineer should be able to get the project completed within the specified schedule and budget while still maintaining the quality.
- **Application modeling skills**: A software engineer should have modeling skills including data, process, and object modeling with the help of various notations.

In addition to the above mentioned qualities, a software engineer should have good communication and interpersonal skills. Moreover, knowledge of object-orientation, quality concept, international organization of standardization (ISO standards), and capability maturity model (CMM) are also required.

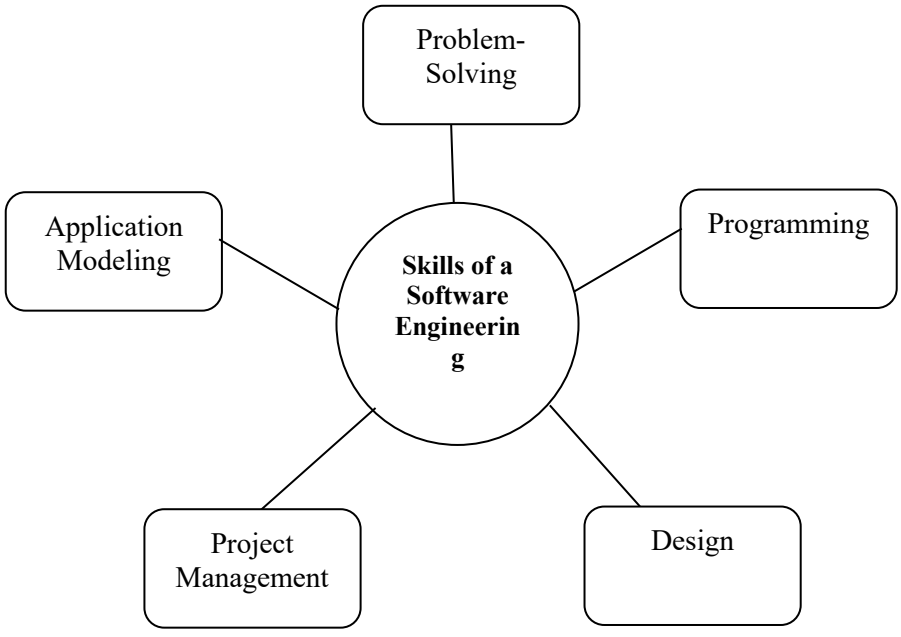

Figure: Skills of a Software Engineer

Software engineers should be committed towards promoting analysis, design, coding, testing, and maintenance of a software as a beneficial and esteemed profession. According to the recommendations

made by IEEE-CS/ACM *Joint Task Force on Software Engineering Ethics and Professional Practices* (version 5.2), the software engineers should adhere to the following principle.

- **Public**: Software engineers shall act consistently with the public interest.
- **Client And employer**: Software engineers shall act in manner that is in the best interests of their client and employer, consistent with public interest.
- **Product**: Software engineers shall ensure their products and related modifications meet the highest professional standards possible.
- **Judgement:** Software engineers shall maintain integrity and independence in their professional judgement.
- **Management:** Software engineers shall subscribe to and promote an ethical approach to the management of software development and maintenance
- **Profession:** Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
- **Colleagues:** Software engineers shall be fair to and supportive of their colleagues.
- **Self:** Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

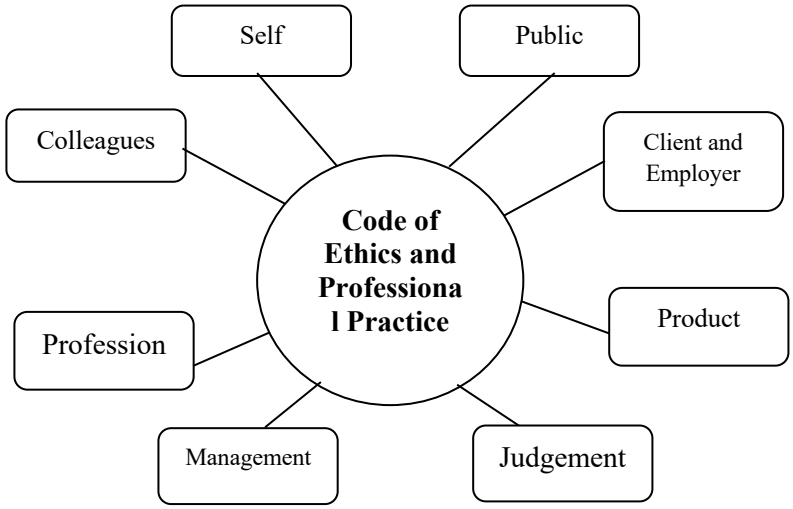

Figure: Code of Ethics and Professional Practice

Software Development Life Cycle (SDLC)

A software system, right from the inception of an idea, to its implementation and delivery to a customer, undergoes through various development and evolution phases. Software is said to have a **Life Cycle** composed of several phases. Each phase has well defined starting and ending points, with clearly identifiable deliverable to the next phase. Each phase may have certain documents, which help to keep track of all activities, procedures, inputs, and outputs etc, associated with that phase of the project. **Software Development Life Cycle (SDLC)** or **software life cycle** is a sequence of activities carried out by **analyst, designers**, and **users** to develop and implement an information system. These activities are carried out in different stages.

- **Analyst** studies the requirements of a customer or user and defines a problem domain. He identifies needs of an organization to determine how people, method and computer technology can best accomplish improvement of the business.

- **Designer** designs the system in terms of the structure of the database, screens, forms and reports. He also determines the hardware and software requirements for the system to be developed.
- **User** is the one who uses the system.

The entire Software Development Life Cycle can be broadly divided into 7 phases, which are:

- **Feasibility Study-** In this phase, we assess whether or not a project should be undertaken. This stage involves defining the problem and fixing up its boundaries. At the end of this stage, the design and development team becomes clear with the project objectives and their work purview.
- **Requirement Analysis-** In this phase the users' requirements are studied and analyzed. The technical development team works with the customers and system end-users to identify the application domain, functions, services, performance capabilities, hardware constraints, related to the system to be developed.
- **System Design-** Is the phase where a new system is designed according to the needs of the user. It is the phase, which finds a solution for the given problem. This is the phase where the specifications of each and every component of the project are laid down.
- **Development-** This is the phase where the system is actually developed. The whole of design phase is built an implemented in this phase.
- **Testing-** This is most crucial phase where the system is judged for all its worth. The system is tested under all kinds of adverse situations and environments to test its performance, reliability and robustness. During this phase entire project functionality is tested with all of its units integrated. The system is tested with test data. During this phase, the developed system is reviewed against each and every Customer Requirement Specification. The developed system should be able to address all of user's needs and its functionality should resolve the client's problem completely.
- **Implementation-**This is the process in which the developed system is handled over to the client. The old system is dispensed, new system is put into operation for use and all the personnel are trained to manage and maintain the new system.
- **Software Maintenance-** This is the phase where in the development team maintains the system for the client. It includes adding enhancements, improvements, updates to newer versions and not just correction of errors and emergency fixes of a system breakdown.

Below figure displays schematic view of phases in SDLC:

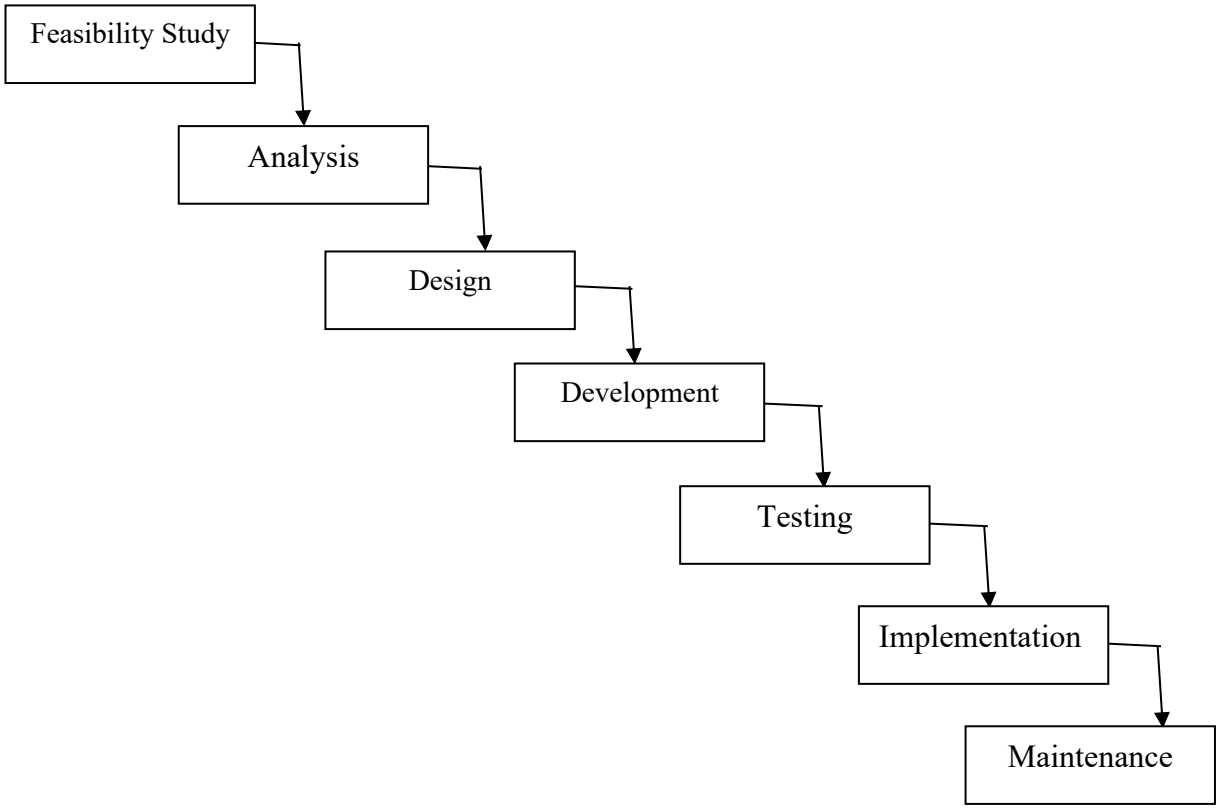

Figure: Schematic views of phases in **SDLC**

Phased Development Process

A development process consist of various phases, each phase ending with a defined output. The phases are performed in an order specified by the process model being followed. The main reason for having a phased process is that it breaks the problem of developing software into successfully performing a set of phases, each handling a different concern of software development. This ensures that the cost of development is lower than what it would have been if the whole problem was tackled together. Furthermore, a phased process allows proper checking for quality and progress at some defined points during the development (end of phases). Without this, one would have to wait until the end to see what software has been produced. Clearly, this will not work for large systems. Hence, for managing the complexity, project tracking, and quality, all the development processes consist of a set of phases. A phased development process is central to the software engineering approach for solving this software crisis.

Various process models have been proposed for developing software. In fact each organization that follows a process has its own version. We will discuss some of the common models in the next chapter. The different processes can have different activities. However, in general, we can say that any problem solving in software must consist of these activities: requirement specification for understanding and clearly stating the problem, design for deciding a plan for a solution, coding for implementing the planned solution, and testing for verifying the programs. 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 a couple of years and where many people are involved in development, performing these activities implicitly without proper documentation and representation will clearly not work, and each of these four problem solving activities has to be done formally. In fact, for large systems, each activity can itself 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.

Table: Building Bridge and Corresponding SDLC Phase

Phase	Building Bridge	SDLC Phase
Formulate the problem by understanding the nature and general requirements of the problem	Understand the locations where the bridge is be built; estimate the traffic that would pass on from the bridge along with its size requirements such as height and width; and so on.	Requirement Analysis
Defining the problem precisely	Specify the exact location where the bridge is to be built along with its size and type	Requirements Specifications
Detailing the solution to the problem	Determine exact configuration, size of cables and beams, and developing blueprints for the bridge	Software design
Implementing	Correspond to actual building of the bridge	Software Coding
Checking	Specify load, pressure, endurance, and robustness of the bridge	Software Testing
Maintaining	Specify the essential repairs that are to be made such as repainting, repaving, etc	Software Maintenance

Waterfall Model

In the waterfall model (also known as the **classical life cycle model**), the development of software proceeds linearly and sequentially from requirement analysis to design, coding, testing, implementation and maintenance. Thus, this model is also known as **linear sequential model.**

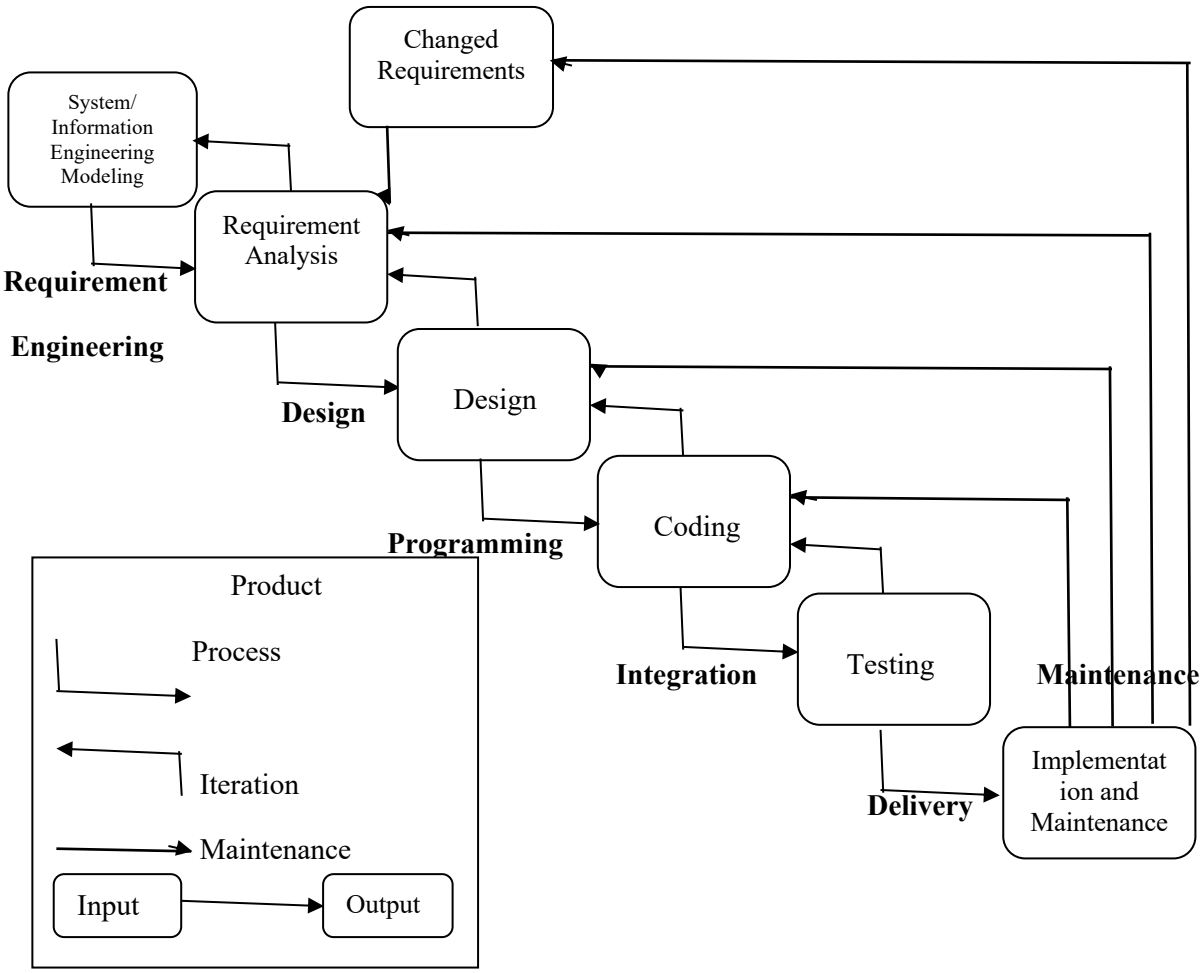

Figure: The Waterfall Model

This model is simple to understand and represents processes which are easy to manage and measure. The waterfall model comprises different phases and each phase has its distinct goal. Figure 3.4.1 shows that after the completion of one phase, the development of software moves to the nest phase. Each phase modifies the intermediate product to develop a new product as an output. The new product becomes the input of the next process.

As stated earlier, the waterfall model comprises several phases, which are listed below.

- **System/information engineering modeling:** This phase establishes the requirements for all parts of the system. Software being a part of the larger system, a subset of these requirements is allocated to it. This system view is necessary when software interacts with other parts of the system including hardware, databases, and people. System engineering includes collecting requirements at the system level while information engineering includes collecting requirements at a level where all decisions regarding business strategies are taken.
- **Requirement analysis and specification phase:** The goal of this phase is to understand the exact requirements of the customer and to document them properly. This activity is usually

executed together with the customer, as the goal is to document all functions, performance and interfacing requirements for the software. The requirements describe the "what" of a system, not the "how". This phase produces a large document, written in a natural language, contains a description of what the system will do without describing how it will be done. The resultant document is known as software requirement specification (SRS) document.

Note: The **SRS** document may act as contract between the developer and customer. If developer fails to implement full set of requirements, it may amount to failure to implement the contracted system.

- **Design phase:** The SRS document is produced in the previous phase, which contains the exact requirements of the customer. The goal of this phase is to transform the requirements specification into a structure that is suitable for implementation in some programming language. Here, overall software architecture is defined, and the high level and detailed design work is performed. This work is documented and kwon as software design description (SDD) document. The information contained in the SDD should be sufficient to begin the coding phase.
- **Coding:** This phase emphasizes translation of design into a programming language using the coding style and guidelines. The programs created should be easy to read and understand. All the programs written are documented according to the specification.
- **Testing:** This phase ensures that the software is developed as per the user's requirements. Testing is done to check that the software is running efficiently and with minimum errors. It focuses on the internal logic and external functions of the software and ensures that all the statements have been exercised (tested). Note that testing is a multistage activity, which emphasizes verification and validation of the software.
- **Operation and maintenance phase:** Software maintenance is a task that every development group has to face, when the software is delivered to the customer's site, installed and is operational. Therefore, release of software inaugurates the operation and maintenance phase of the life cycle. The time spent and effort required to keep the software operational after release is very significant. Despite the fact that it is a very important and challenging task; it is routinely the poorly managed headache that nobody wants to face. Software maintenance is a very broad activity that includes error correction, enhancement of capabilities, deletion of obsolete capabilities, and optimization. The purpose of this phase is to preserve the value of the software over time. This phase may span for 5 to 50 years whereas development may be 1 to 3 years.

  This model is easy to understand and reinforces the notion of "define before design" and "design before code". This model expects complete and accurate requirements early in the process, which is unrealistic. Working software is not available until relatively late in the process, thus delaying the discovery of serious errors. It also does not incorporate any kind of risk assessment.

Advantages

1. Relatively simple to understand.
2. Each phase of development proceeds sequentially.
3. Allows managerial control where a schedule with deadlines is set for each stage of development.
4. Helps in controlling schedules, budgets, and documentation.

Disadvantages

1. Requirements need to be specified before the development proceeds.
2. The changes of requirements in later phases of the waterfall model cannot be done. This implies that once the software enters the testing phase, it becomes difficult to incorporate changes at such a late phase.
3. No user involvement and working version of the software is available when the software is being developed.
4. Does not involve risk management.
5. Assumes that requirements are stable and are frozen across the project span.

Prototyping Model

The prototyping model suggests that before carrying out the development of the actual software, a working prototype of the system should be built. A prototype is a toy implementation of the system. A prototype usually exhibits limited functional capabilities, low reliability, and inefficient performance compared to the actual software. A prototype is usually built using several shortcuts might involve using inefficient, inaccurate, or dummy functions. The shortcut implementation of a function, for example, may produce the desired results by using a table look-up instead of performing the actual computations. A prototype usually turns out to be a very crude version of the actual system.

There are several uses of a prototype. An important purpose is to illustrate the input data formats, massages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs. In this regard, the prototype model is very useful in developing the Graphical User Interface (GUI) part of a system. It becomes much easier for the user to form his opinion by experimenting with a working model, rather than trying to imagine the working of a hypothetical system.

Another important situation where the prototyping model can be used is when the technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with product development. Often, major design decisions depends depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. in such circumstances, a prototype may be the best or the only way to resolve the technical issues. The third reason for developing a prototype is that it is impossible to "get it right" the first time, and one must plan to throw away the first product in order to develop a good product later, as advocated by Brooks [1975].

The prototyping model of software development is shown in Figure 3.4.2. In this model, product development starts with an initial requirements gathering phase. A quick design is carried out and the prototype is built. The developed prototype is submitted to the customer for his evaluation. Based on the customer feedback, the requirements are refined and the prototype is suitably modified. This cycle of obtaining customer feedback and modifying the prototype continues till the customer approves the prototype. The actual system is developed using the iterative waterfall approach. However, in the prototyping model of development, the requirements analysis and specification phase becomes an animated requirements specification.

The code for the prototype is usually thrown away. However, the experience gathered from developing the prototype helps a great deal in developing the actual system. Therefore, even though the construction of a working prototype might involve additional cost, for systems with unclear customer requirements and for systems with resolved technical issues, the overall development cost might turn out to be lower

in the prototype model than that of an equivalent system developed using the iterative waterfall model. By constructing the prototype and submitting it for user evaluation, many customer requirements get properly defined requests from the customer and the associated redesign costs.

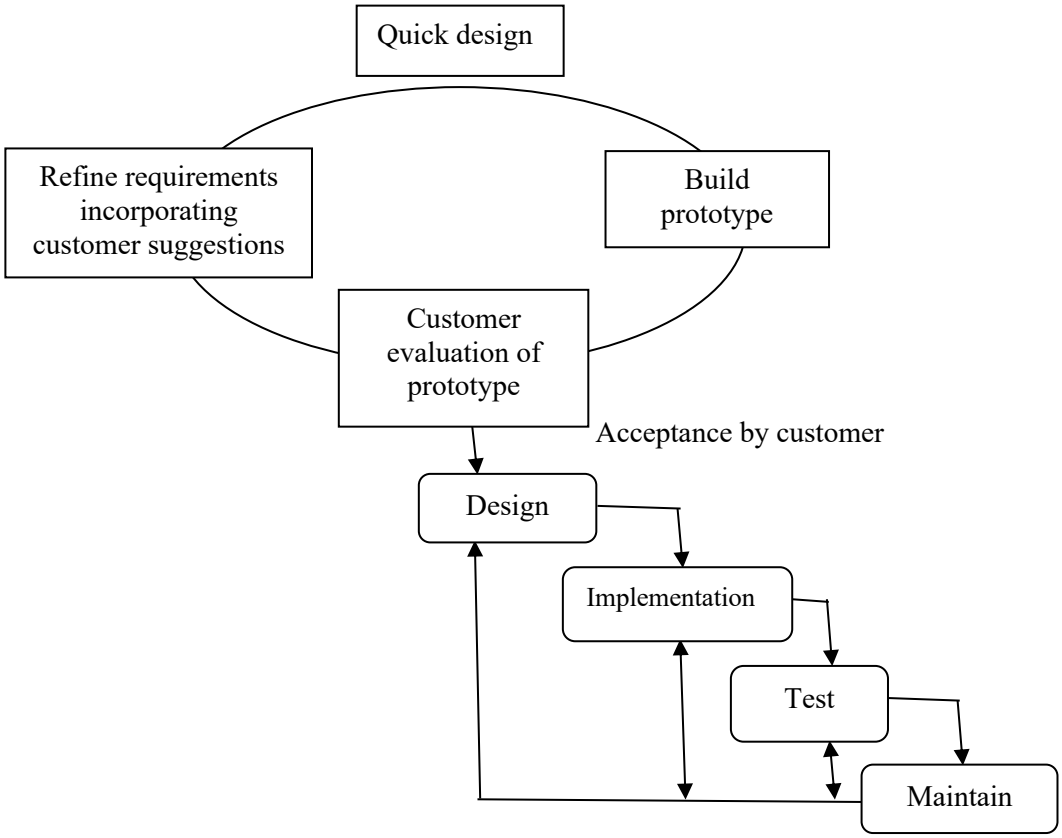

Figure: Prototyping model of software development

Advantages

1. Provides a working model to the user early in the process, enabling early assessment and increasing user's confidence.
2. The developer gains experience and insight by developing a prototype thereby resulting in better implementation of requirements.
3. The prototyping model serves to clarify requirements, which are not clear, hence reducing ambiguity and improving communication between the developers and users.
4. There is a great involvement of users in software development. Hence, the requirements of users are met to the greatest extent.
5. Helps in reducing risks associated with the software.

Disadvantages

1. If the user is not satisfied by the developed prototype, then a new prototype is developed. This process goes on until a perfect prototype is developed. Thus, this model is time consuming and expensive.
2. The developer loses focus of the real purpose of prototype and hence, may compromise with the quality of the software. For example, developers may use some inefficient algorithms or inappropriate programming languages while developing the prototype.

3. Prototyping can lead to false expectations. For example, a situation may be created where the user believes that the development of the system is finished when it is not.
4. The primary goal of prototyping is speedy development; the system design can suffer as it is developed in series without considering integration of all other components.

The Rapid Application Development Model (RAD) Model

This model is an incremental process model and was developed by IBM in the 1980s and described in the book of James Martin entitled "Rapid Application Development". Hence, use involvement is essential from requirement phase to delivery of the product. The continuous user participation ensures the involvement of user's expectations and perspective in requirements elicitation, analysis and design of the system.

The process is started with building a rapid prototype and is given to user for evaluation. The user feedback is obtained and prototype is refined. The process continues, till the requirements are finished. We may use any grouping technique (like FAST, QFD, Brainstorming sessions; for details refer chapter 3) for requirements elicitation. Software requirement and specification (SRS) and design documents are prepared with the association of users.

There are four phases in this model and these are shown in figure.

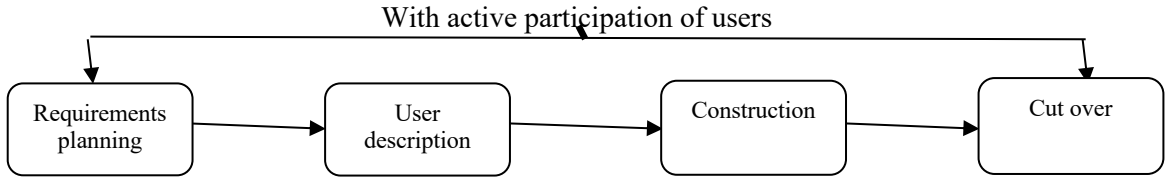

Figure: RAD Model

1. **Requirements planning phase:** Requirements are captured using any group elicitation technique. Some techniques are discussed in chapter 4. Only issue is the active involvement of users for understanding the project.
2. **User description:** Joint teams of developers and users are constituted to prepare understand and review the requirements. The team may use automated tools to capture information from the other users.
3. **Construction phase:** This phase combines the detailed design, coding and testing phase of waterfall model. Here, we release the product to customer. It is expected to the code generators, screen generators and the other types of productivity tools.
4. **Cut over phase:** This phase incorporates acceptance testing by the users, installation of the system, and user training.

In this model, quick model initial views about the product are possible due to delivery of rapid prototype. The development time of the product may be reduced due to use of powerful development tools. It may use CASE tools and frameworks to increase productivity. Involvement of user may increase the acceptability of the product.

If user cannot be involved throughout the life cycle, this may not be an appropriate model. Development time may not be reduced very significantly, if reusable components are not available. Highly specialized and skilled developers are expected and such developers may not be available very easily. It may not be effective, if system cannot be properly modularised.

Advantages

1. Deliverables are easier to transfer as high-level abstractions, scripts, and intermediate codes are used.
2. Provides greater flexibility as redesign is done according to the developer.
3. Results in reduction of manual coding due to code generators and code reuse.
4. Encourages user involvement.
5. Possibility of lesser defects due to prototyping in nature.

Disadvantages

1. Useful for only larger projects.
2. RAD projects fail if there is no commitment by the developers or the users to get software completed on time.
3. Not appropriate when technical risks are high. This occurs when the new application utilizes new technology or when new software requires a high degree of interoperability with existing system.
4. As the interests of users and developers can diverge from single iteration to next, requirements may not converge in RAD model.

V model

The V model is useful in every phase of the software development life cycle. This model determines the complex relationship between each phase of the software development and ensures that each phase of software development is associated with testing. Various functions performed by V model are listed below.

- It emphasizes that testing occurs in every phase of software development and does not occur only after the coding is completed.
- It determines the software development process within organization.
- It determines the activities and the results to be produced in the software development.
- It describes the products to be created during the software project.

The V model is divided into two branches: **left and right.** The **left** branch analyses and determines the requirements of the software to be developed. The **right** branch, on the other hand, includes the testing activities. The left and right branches of this model work concurrently. This implies that a relationship is established between both the branches as shown in figure.

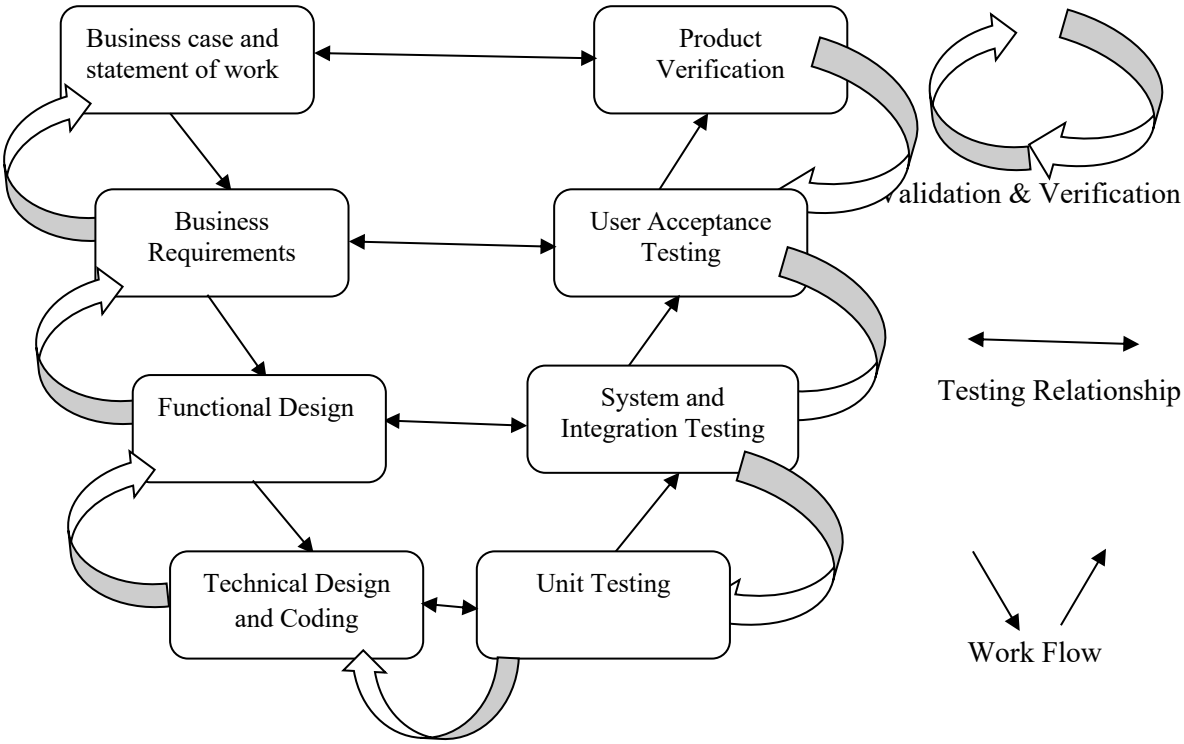

Figure: The V Model

The interrelationship between the branches of V model can be determined by the following points.

- Unit testing verifies the technical design. It individually tests units to verify that they (units) are functioning according of the requirements
- The integration and system testing verifies the functional design of the program to ensure that the functional design is implemented correctly.
- The requirements of the business are validated at the user end with the help of acceptance testing. Acceptance testing is used when the final software is developed. It verifies that the software is developed. It verifies that the software has been developed according to the user's requirements.
- Last of all, the production verification is carried out.

One of the key aspects of the V model is that verification and validation are performed simultaneously in both the branches. It is essential to link the left branch with right branch as the left side of V model is executed to correct problems, which are encountered during verification and validation (the right side).

Various advantages and disadvantages associated with the V model are listed in below.

Advantages

1. Covers all functional areas.
2. Contains instructions and recommendations, which provide a detailed explanation of problems involved.
3. Emphasizes the significance of testing and ensures that testing is planned.

Disadvantages

1. The processes are institutionalized during the project and when the project is finished, they are abolished.

Spiral Model

This is a recent that has proposed by Boehm [Boe88]. As the name suggests, the activities in this model can be organized like a spiral that has many cycles. The radial dimension represents the cumulative cost incurred in accomplishing the steps done so far, and the angular dimension represents the progress made in completing each cycle of the spiral. Each cycle in the spiral begins with the identification of objectives for that cycle, the different alternatives that are possible for achieving the objectives, and the constraints that exits. This is the first quadrant of the cycle (upper-left-quadrant). The next step in the cycle is to evaluate these different alternatives based on the objectives and constraints. The focus of evaluation in this step is based on the risk perception for the project. Risks reflect the chances that some of the objectives of the project may not be met. Risk management will be discussed in more detail later in the book. The next step is to develop strategies that resolve the uncertainties and risks. This step may involve activities such as benchmarking, simulation, and prototyping. Next, the software is developed, keeping in mind the risks. Finally the next stage is planned.

The development step depends on the remaining risks. For example, if performance or user interface risks are considered more important than the program development risks, the next step may be an evolutionary development that involves developing a more detailed prototype for resolving the risks. On the other hand, if the program development risks dominate and the previous prototype have resolved all the user-interface and performance risks, the next step will follow the basic waterfall approach.

The risk driven nature of the spiral model allows it to accommodate any mixture of a specification-oriented, prototype-oriented, simulation-oriented, or some other type of approach. An important feature of the model is that each cycle of the spiral is completed by a review that covers all the products developed during that cycle, including plans for the next cycle. The spiral model works for development as well as enhancement projects.

In a typical application of the spiral model, might start with an extra round zero, in which the feasibility of the basic project objectives is suited. These project objectives may or may not lead to a development/enhancement project. Such high-level objectives include increasing the efficiency of code generation of a compiler, producing a new full-screen text editor and developing an environment for improving productivity. The alternatives considered in this round are also typically very high-level, such as whether the organization should go for in-house development, or contract it out, or buy an existing product. In round one, a concept of operation might be developed. The objectives are stated more precisely and quantitatively and the cost and other constraints are defined precisely. The risks here are typically whether or not the goals can be met within the constraints. The plan for the next phase will be developed, which will involve defining separate activities for the project. In round two the top-level requirements are developed. In succeeding rounds the actual development may be done.

This is a relatively new model; it can encompass different development strategies. In addition to the development activities, it incorporates some of the management and planning activities into the model. For high-risk project this might be a preferred model.

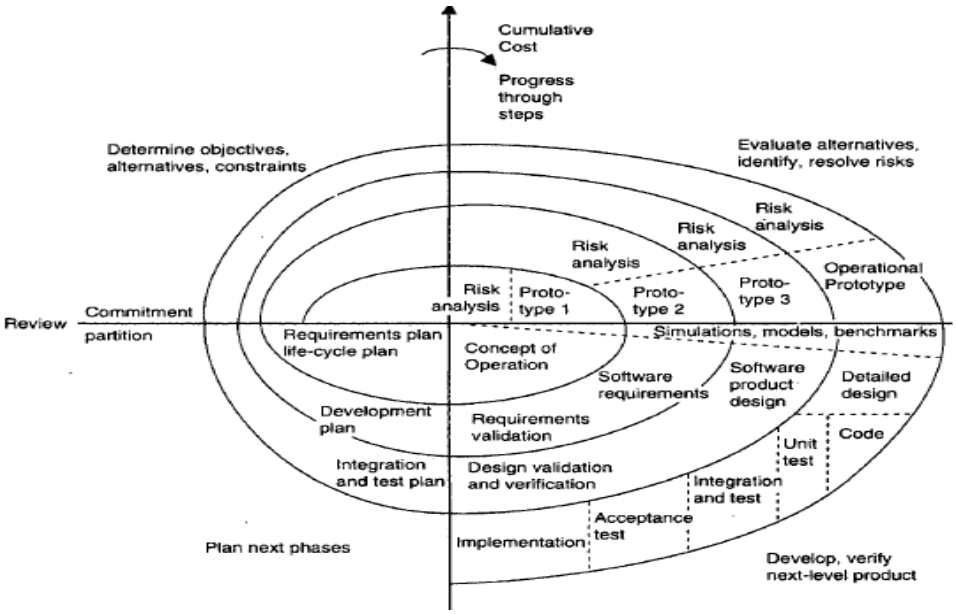

Figure: Spiral Model

Advantages

1. Avoids the problem resulting in risk-driven approach in the software.
2. Specifies a mechanism for software quality assurance activities.
3. Is utilized by complex and dynamic projects.
4. Re-evaluation after each step allows changes in user perspectives, technology advances, or financial perspectives.
5. Estimation of budget and schedule gets realistic as the work progresses.

Disadvantages

1. Assessment of project risks and its resolution is not an easy task.
2. Difficult to estimate budget and schedule in the beginning as some of the analysis is not done until the design of the software is developed.

Selection of a Life Cycle Model

The selection of a suitable model is based on the following characteristics/categories:

1. Requirements
2. Development team
3. Users
4. Project type and associated risk

Characteristics of Requirements

Requirements are very important for the selection of an appropriate model. There are number of situations and problems during requirements capturing and analysis. The details are given in table:

Table: Selection of a model on characteristics of requirements

Requirements	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Are requirements easily understandable and defined	YES	NO	NO	NO	NO	YES
Do we change requirements quite often?	NO	YES	NO	NO	YES	NO
Can be define requirements early in the cycle?	YES	NO	YES	YES	NO	YES
Requirements are indicating a complex system to be built	NO	YES	YES	YES	YES	NO

Status of Development Team

The status of development team in terms of availability, effectiveness, knowledge, intelligence team work etc., is very important for the success of the project. If we know above mentioned parameters and characteristics of the team, then we may choose an appropriate life cycle model for the project. Some of the details are given in table:

Table: Selection based on status of development team

Development	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Less experience on similar projects	NO	YES	NO	NO	YES	NO
Less domain knowledge (new to the technology)	YES	NO	YES	YES	YES	NO
Less experience on tools to be used	YES	NO	NO	NO	YES	NO
Availability of training if required	NO	NO	YES	YES	NO	YES

Involvement of Users

Involvements of users increment the understandability of the project. Hence user participation, if available, plays a very significant role in the selection of an appropriate life cycle model. Some issues are discussed in table.

Table: Selection based on user's participation

Involvement of users	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
User involvement in all phases	NO	YES	NO	NO	NO	YES
Limited user participation	YES	NO	YES	YES	YES	NO
User have no previous experience of participation in similar	NO	YES	YES	YES	YES	NO
Users are experts of problem domain	NO	YES	YES	YES	NO	YES

Type of Project and Associated Risk

Very few models incorporate risk assessment. Project type is also important for the selection of a model. Some issues are discussed in table.

Table: Selection based on type of project with associated risk

Project type and risk	Waterfall	Prototype	Iterative enhancement	Evolutionary development	Spiral	RAD
Projects is the enhancement of the existing system	NO	NO	YES	YES	NO	YES
Funding is stable for the project	YES	YES	NO	NO	NO	YES
High reliability requirements	NO	NO	YES	YES	YES	NO
Tight project schedule	NO	YES	YES	YES	YES	YES
Use of reusable components	NO	YES	NO	NO	YES	YES
Are resources (time, money people etc.) scarce?	NO	YES	NO	NO	YES	NO

An appropriate model may be selected based on options given in four tables. Firstly, we have to answer the questions presented for each category by circling a yes or on in each table. Rank the importance of each category, or question within the category, in terms of the project for which we want to select a model. The total number of circled responses for each column in the tables decide an appropriate model. We may also use the category ranking to resolve the conflict between models if the total in either case is close or the same.