UNIT - I

Basic concepts in software engineering and software project management : Basic concepts: abstraction versus decomposition, evolution of software engineering techniques, Software development life cycle (SDLC) models: Iterative waterfall model, Prototype model, Evolutionary model, Spiral model, RAD model, Agile models

Software project management: project planning, project estimation, COCOMO, Halstead's Software Science, project scheduling, staffing, Organization and team structure, risk management, configuration management.

Software and Software Engineering

Software engineering stands for the term is made of **two** words, S**oftware** and E**ngineering.**

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is **an efficient and reliable software product**.

Definitions

IEEE defines **Software Engineering** as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in the above statement.

Fritz Bauer, a German computer scientist, defines software engineering as:

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

1.1 BASIC CONCEPTS IN SOFTWARE ENGINEERING

Software engineering is an engineering approach for software development. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively.

A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes. Software engineering helps to reduce the programming complexity. Software engineering principles use **two** important techniques to reduce problem complexity: ***abstraction and decomposition***.

Abstraction: Abstraction refers to construction of a simpler version of a problem by ignoring the details. The principle of constructing an abstraction is popularly known as **modeling**

The principle of abstraction shown in following fig, it implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem.

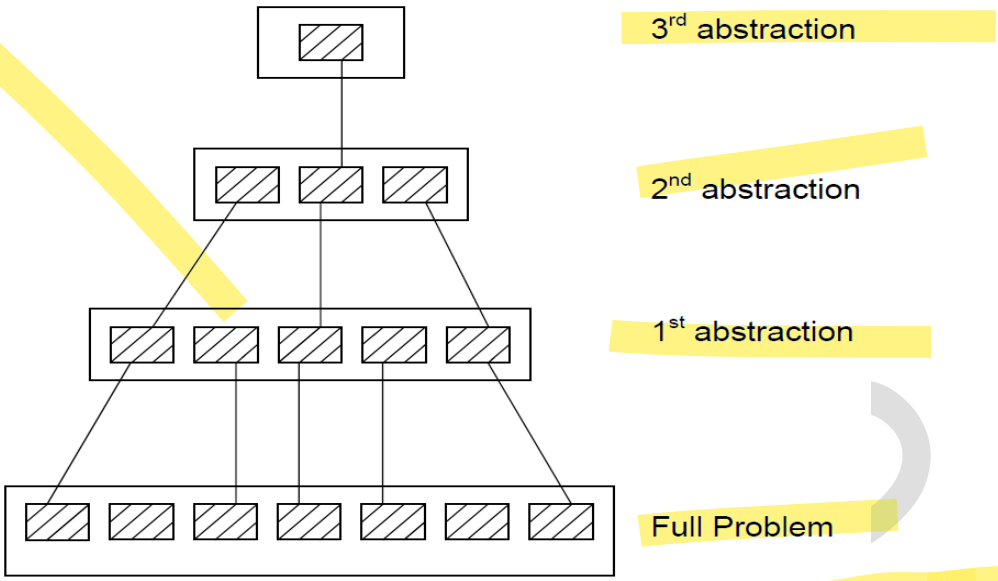


Fig. : A hierarchy of abstraction

Decomposition: Decomposition is another important principle that is available in the set of skills of a software engineer to handle problem complexity. This principle is largely made use by several software engineering techniques to contain the exponential growth of the perceived problem complexity. The decomposition principles are popularly known as the divide and conquer principle.

A good decomposition of a problem as shown in following fig. should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

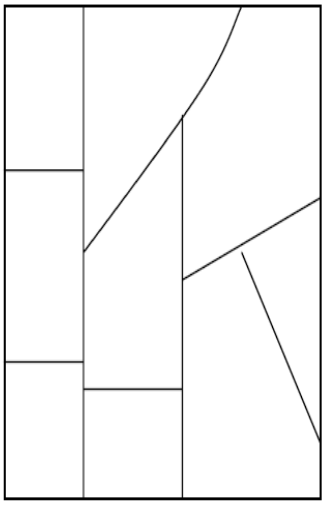


Fig: Decomposition of a large problem into a set of smaller problems.

1.2 Evolution of software engineering techniques

ECDDOA -> refer PPT for full form

Early Computer Programming

Software engineering principles have evolved over the last sixty years in numerous ways. The early programmers used an **ad hoc programming** style. These styles of program development are now variously being referred to as **exploratory, build and fix, and code and fix styles**.

The exploratory programming style is an informal style in the sense that there are no set rules or recommendations that a programmer has to follow. The exploratory development style gives complete freedom to the programmer to choose the activities using which to develop software.

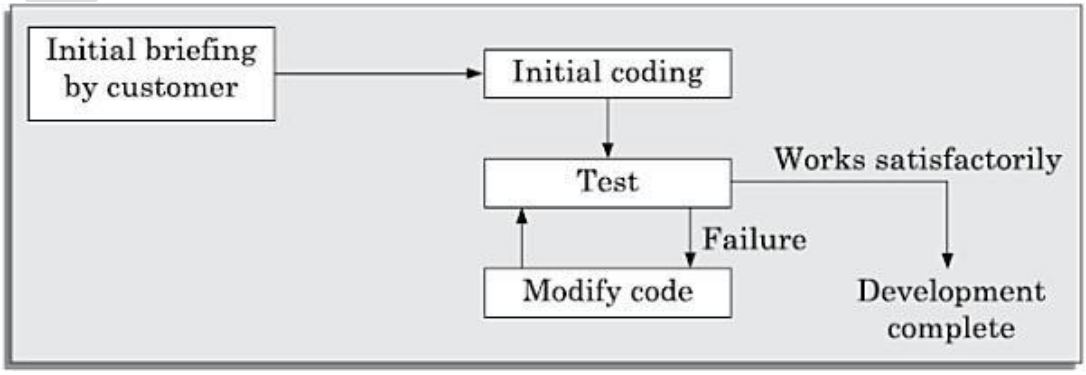


Figure: Exploratory program development

High-level Language Programming

Computers became faster with the introduction of the **semiconductor technology** in the early 1960s.. At this time, high-level languages such as FORTRAN, ALGOL, and COBOL were introduced which contributed significantly to develop software and helped programmers to write larger programs.

Control Flow-based Design

As the size and complexity of programs kept on increasing, the exploratory programming style proved to be insufficient. Programmers found it increasingly difficult not only to write cost-effective and correct programs, but also to understand and maintain programs written by others. To cope up with this problem, experienced programmers advised other programmers to pay particular attention to the design of program's **control flow structure**. A program's control flow structure indicates the sequence in which the program's instructions are executed. In order to help develop programs having good control flow structures, the **flow charting technique** was developed.

In late 1960s, it was found that the "GOTO" statement causes difficulties in control structure programming. In order to restrict the use of GO TO statements, structured programming methodology was introduced as a part of which several languages such as PASCAL, MODULA, C, etc., became available which were specifically designed to support **structured programming**. These programming languages facilitated writing **modular programs** and programs having good control structures.

Data Structure-oriented Design

Computers became even more powerful with the advent of **integrated circuits (ICs)** in the early 1970s. These could now be used to solve more complex problems. Software developers were tasked to develop larger and more complicated software. The control flow-based program development techniques could not be used satisfactorily any more to write those programs, and more effective program development techniques were needed. It was soon discovered that while developing a program, it is much more important to pay attention to the design of the important **data structures of the program** than to the design of its control

structure. Keeping this in view, utmost importance was given to the Data structure-oriented design techniques using which the code structure is designed.

Data Flow-oriented Design

As computers became faster and more powerful with the introduction of **very large scale integrated (VLSI) Circuits,** the more effective software designing techniques such as **data flow-oriented techniques** which included ***data flow diagrams*** **(DFD)** were proposed to solve further challenging problems.

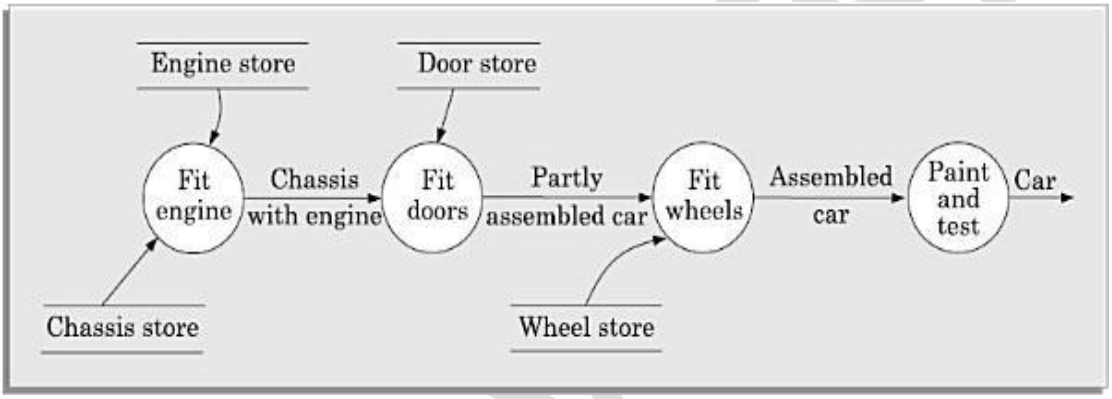


Figure: Data flow model of a car assembly plant

Object-oriented Design

Data flow-oriented techniques evolved into object-oriented design (OOD) techniques in the late 1970s. Object-oriented design technique is an intuitively appealing approach, where the natural objects (such as employees, pay-roll-register, etc.) relevant to a problem are first identified and then the relationships among the objects such as composition, reference, and inheritance are determined. Each object essentially acts as a data hiding (also known as data abstraction) entity.

The evolution of software designing has reached a new era with the rapid invention of Aspect-oriented programming, Client- server based design, and embedded software design techniques which pictorially summarized in the following figure.

Figure : Evolution of software design techniques.

Other Developments

It can be seen that remarkable improvements to the prevalent software design technique occurred almost every passing decade. The improvements of new techniques include life cycle models, specification techniques, project management techniques, testing techniques, debugging techniques, quality assurance techniques, software measurement techniques, computer aided software engineering (CASE) tools, etc.

1.3. Software development life cycle (SDLC) models

SDLC is a process that defines the various stages involved in the development of software for delivering a high-quality product. It has defined its phases from Inception Phase to Maintenance Phase as a series of identifiable stages that a software product undergoes during its life time. In simple words, we can define an SDLC as follows:

An SDLC graphically depicts the different phases through which a software evolves. It is usually accompanied by a textual description of the different activities that need to be carried out during each phase.

Process versus methodology

A software development process has a much broader scope as compared to a software development methodology. A process usually describes all the activities starting from the inception of a software to its maintenance and retirement stages, or at least a chunk of activities in the life cycle. It also recommends specific methodologies for carrying out each activity. A methodology, in contrast, describes the steps to carry out only a single or at best a few individual activities.

A software development life cycle (SDLC) model (also called software life cycle model and software development process model) describes the following stages in its life cycle.

- Feasibility study (1. OPERATIONAL, 2. ECONOMY 3. TECHNICAL)
- Requirements analysis and specification (SRS),
- Design (model or Architecture) (flowchart, ER-diagram, DFD, OOAD etc)
- Coding,
- Testing and Deployment
- Maintenance.

A few important and commonly used models :

- Classical waterfall model
- Iterative waterfall,
- V MODEL
- Evolutionary,
- Prototyping, and RAD MODEL
- Spiral Model
- AGILE MODEL

1.3.1. **Classical Waterfall Model**

The ***waterfall model***, sometimes called the ***classic life cycle***, suggests a systematic, sequential approach.

The classical waterfall model divides the life cycle into a set of phases as shown in the following figure. It can be understood from the figure that the life cycle of a software development process resembles a multi-level waterfall.

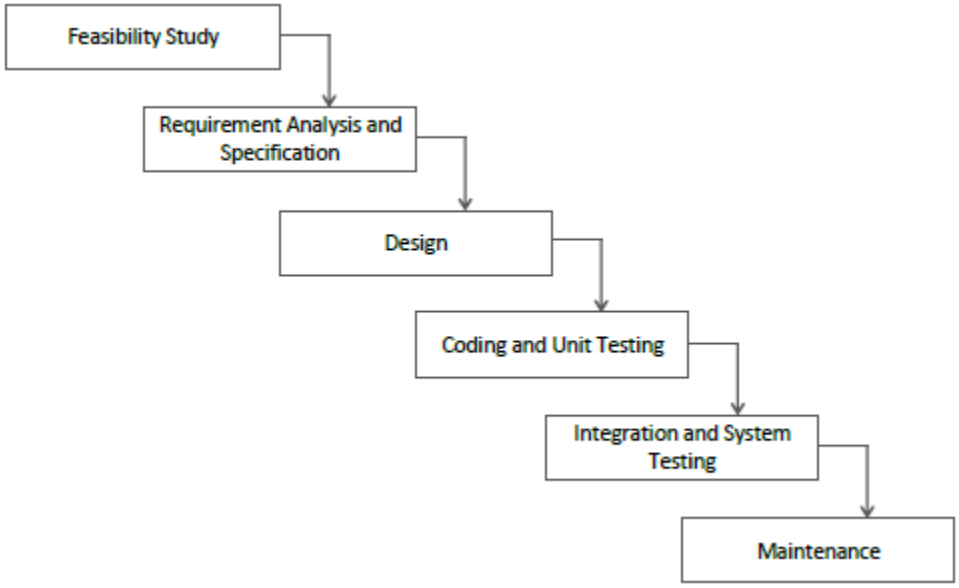


Figure : Classical waterfall model

Phases of the classical waterfall model

The different phases of the classical waterfall model have been shown in above Figure, the different phases are—**feasibility study, requirements analysis and specification, design, coding and unit testing, integration and system testing, and maintenance**. The phases starting from the feasibility study to the integration and system testing phase are known as the **development phases**. A software is developed during the development phases, and at the completion of the development phases, the software is delivered to the customer. After the delivery of software, customers start to use the software signalling the commencement of the **operation phase**. As the customers start to use the software, changes to it become necessary on account of bug fixes and feature extensions, causing maintenance works to be undertaken. Therefore, the last phase is also known as the **maintenance phase** of the life cycle.

Feasibility study

The main focus of the feasibility study stage is to determine whether it would be financially and technically feasible to develop the software. The feasibility study involves

carrying out several activities such as collection of basic information relating to the software such as the different data items that would be input to the system, the processing required to be carried out on these data, the output data required to be produced by the system, as well as various constraints on the development. These collected data are analyzed to perform at the following:

- **Development of an overall understanding of the problem:** It is necessary to first develop an overall understanding of what the customer requires to be developed.
- **Formulation of the various possible strategies for solving the problem:** In this activity, various possible high-level solution schemes to the problem are determined. For example, solution in a client-server framework and a standalone application framework may be explored.
- **Evaluation of the different solution strategies:** The different identified solution schemes are analyzed to evaluate their benefits and shortcomings. Such evaluation often requires making approximate estimates of the resources required, cost of development, and development time required. The different solutions are compared based on the estimations that have been worked out. Once the best solution is identified, all activities in the later phases are carried out as per this solution. At this stage, it may also be determined that none of the solutions is feasible due to high cost, resource constraints, or some technical reasons. This scenario would, of course, require the project to be abandoned.

Requirements analysis and specification

The aim of the requirements analysis and specification phase is to understand the exact requirements of the customer and to document them properly. This phase consists of **two** distinct activities, namely **requirements gathering and analysis**, and **requirements specification**.

- **Requirements gathering and analysis:** The goal of the requirements gathering activity is to collect all relevant information regarding the software to be developed from the customer with a view to clearly understand the requirements. The goal of the requirements analysis activity is to weed out the incompleteness and inconsistencies in these gathered requirements.

- **Requirements specification:** After the requirement gathering and analysis activities are complete, the identified requirements are documented. This is called a **software requirements specification (SRS)** document. The SRS document is written using end-user terminology. This makes the SRS document understandable to the customer.

Design

The goal of the design phase is to transform the requirements specified in the SRS document into a structure that is suitable for implementation in some programming language. In technical terms, during the design phase the software architecture is derived from the SRS document. **Two** distinctly different design approaches are popularly being used at present—**the procedural** and **object-oriented design** approaches.

- **Procedural design approach:** This traditional design technique is based on the **data flow-oriented design** approach. It consists of **two** important activities; **first structured analysis** of the requirements specification is carried out where the detailed structure of the problem is examined. This is followed by a **structured design** step where the results of structured analysis are transformed into the software design.
- **Object-oriented design approach:** In this technique, various objects that occur in the problem domain and the solution domain are first identified and the different relationships that exist among these objects are identified. The object structure is further refined to obtain the detailed design.

Coding and unit testing

The purpose of the coding and unit testing phase is to translate a software design into source code and to ensure that individually each function is working correctly. The coding phase is also sometimes called the **implementation phase**. The end-product of this phase is a set of program modules that have been individually unit tested. The main objective of unit testing is to determine the correct working of the individual modules.

Integration and system testing

Integration testing is carried out to verify that the interfaces among different units are working satisfactorily. On the other hand, the goal of system testing is to ensure that the developed system conforms to the requirements that have been laid out in the SRS document.

System testing usually consists of three different kinds of testing activities:

- **α-testing:** testing is the system testing performed by the development team.
- **β-testing:** This is the system testing performed by a friendly set of customers.
- **Acceptance testing:** After the software has been delivered, the customer performs system testing to determine whether to accept the delivered software or to reject it.

Maintenance

The total effort spent on maintenance of a typical software during its operation phase is much more than that required for developing the software itself. Maintenance is required in the following **three** types of situations:

- **Corrective maintenance:** This type of maintenance is carried out to correct errors that were not discovered during the product development phase.
- **Perfective maintenance:** This type of maintenance is carried out to improve the performance of the system, or to enhance the functionalities of the system based on customer's requests.
- **Adaptive maintenance:** Adaptive maintenance is usually required for porting the software to work in a new environment.

Shortcomings of the classical waterfall model

- No feedback paths
- Difficult to accommodate change requests
- Inefficient error corrections
- No overlapping of phases

1.3.2. Iterative Waterfall Model

The main change brought about by the iterative waterfall model to the classical waterfall model is in the form of providing **feedback paths** from every phase to its preceding phases. The feedback paths introduced by the iterative waterfall model are shown in following Figure. The feedback paths allow for correcting errors committed by a programmer during some phase, as and when these are detected in a later phase.

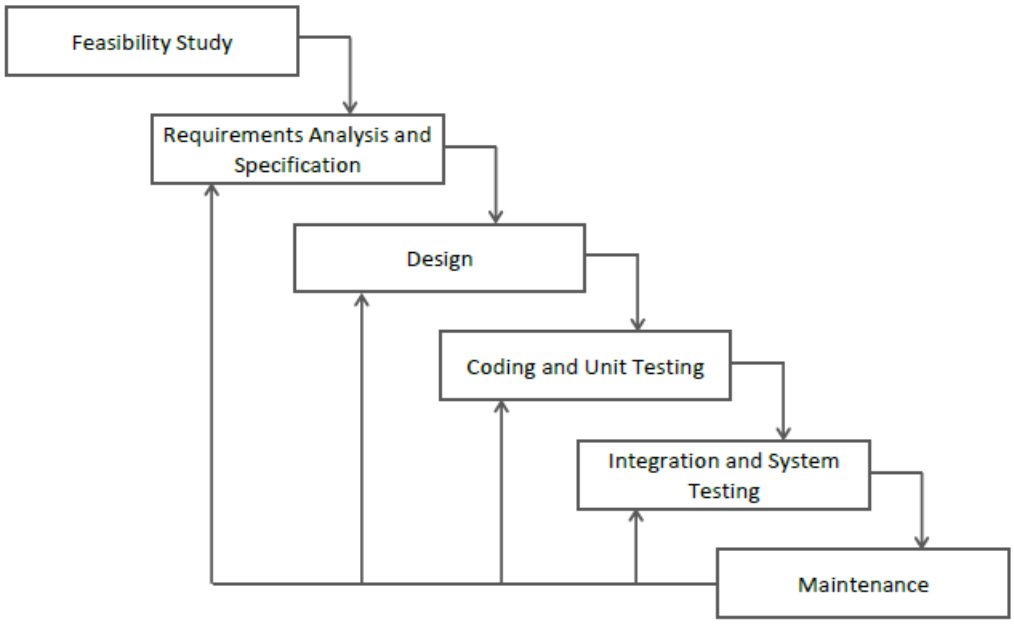


Figure: Iterative waterfall model

Almost all life cycle models are iterative in nature, except the classical waterfall model and the V-model—which are sequential in nature. In a sequential model, once a phase is complete, no work product of that phase is changed later.

Shortcomings of the iterative waterfall model

- Difficult to accommodate change requests
- Incremental delivery not supported
- Phase overlap not supported
- Phase overlap not supported
- Error correction unduly expensive
- Limited customer interactions
- Heavy weight

- No support for risk handling and code reuse

1.3.3. V-Model

V-model is a variant of the waterfall model. As is the case with the waterfall model, this model gets its name from its visual appearance. In this model verification and validation activities are carried out throughout the development life cycle, and therefore the chances bugs in the work products considerably reduce. This model is therefore generally considered to be suitable for use in projects concerned with development of safety-critical software that are required to have high reliability**.**

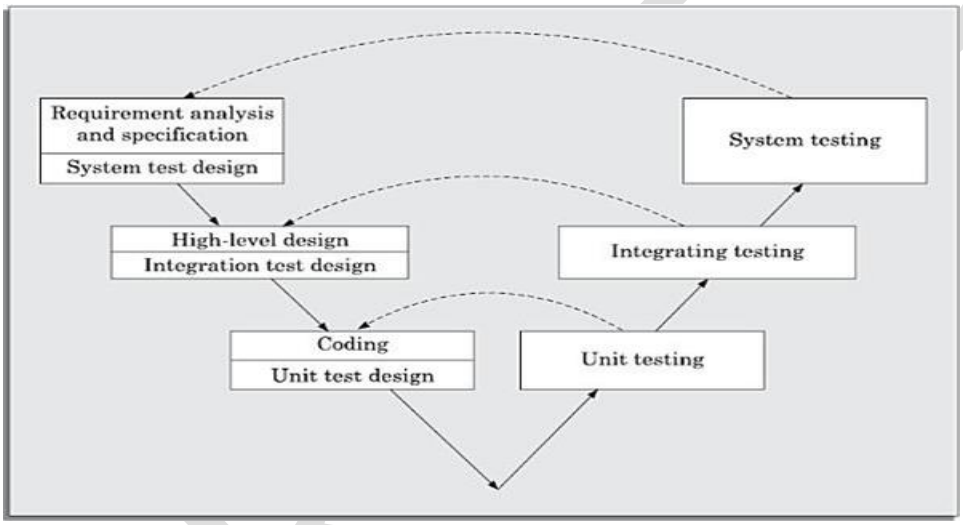


Figure: V-model

As shown in Figure, there are **two** main phases—**development** and **validation** phases. The **left half** of the model comprises the development phases and the **right half** comprises the validation phases.

- In each development phase, along with the development of a work product, test case design and the plan for testing the work product are carried out, whereas the actual testing is carried out in the validation phase. This validation plan created during the development phases is carried out in the corresponding validation phase which have been shown by dotted arcs in Figure.
- In the validation phase, testing is carried out in **three** steps—**unit, integration, and system testing**. The purpose of these three different steps of testing during the

validation phase is to detect defects that arise in the corresponding phases of software development— requirements analysis and specification, design, and coding respectively.

V-model *versus* waterfall model

The V-model can be considered to be an extension of the waterfall model. However, there are major differences between the two. In contrast to the iterative waterfall model where testing activities are confined to the testing phase only, In the V-model testing activities are spread over the entire life cycle. As shown in figure during the requirements specification phase, the system test suite design activity takes place. During the design phase, the integration test cases are designed. During coding, the unit test cases are designed.

1.3.4. Prototyping Model

The prototyping model can be considered to be an extension of the waterfall model. This model suggests building a working ***prototype*** of the system, before development of the actual software. A prototype is a toy and crude implementation of a system. It has limited functional capabilities, low reliability. A prototype can be built very quickly by using several shortcuts. The shortcuts usually involve developing inefficient, inaccurate, or dummy functions.

The prototyping model is advantageous to use for specific types of projects. There are three types of projects for which the prototyping model can be followed to advantage:

- It is advantageous to use the prototyping model for development of the ***graphical user interface*** **(GUI)** part of an application. Through the use of a prototype, it becomes easier to illustrate the input data formats, messages, reports, and the interactive dialogs to the customer. This is a valuable mechanism for gaining better understanding of the customers' needs. In this regard, the prototype model turns out to be especially useful in developing the ***graphical user interface*** **(GUI)** part of a system.
- The prototyping model is especially useful when the exact technical solutions are unclear to the development team. A prototype can help them to critically examine the technical issues associated with product development.
- An important reason for developing a prototype is that it is impossible to "get it right" the first time. The prototyping model can be deployed when development of highly optimized and efficient software is required.

Note: The prototyping model is considered to be useful for the development of not only the GUI parts of a software, but also for a software project for which certain technical issues are not clear to the development team.

Life cycle activities of prototyping model

The prototyping model of software development is graphically shown in following figure. Software is developed through **two** major activities—**prototype construction** and **iterative waterfall-based** software development.

Prototype development: Prototype development starts with an initial requirement gathering phase. A quick design is carried out and a prototype is built. The developed prototype is submitted to the customer for 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.

Iterative development: Once the customer approves the prototype, the actual software is developed using the iterative waterfall approach. In spite of the availability of a working prototype, the SRS document is usually needed to be developed since the SRS document is invaluable for carrying out traceability analysis, verification, and test case design during later phases. However, for GUI parts, the requirements analysis and specification phase become redundant since the working prototype that has been approved by the customer serves as an animated requirements specification.

Strengths of the prototyping model

This model is the most appropriate for projects that suffer from technical and requirements risks. A constructed prototype helps overcome these risks.

Weaknesses of the prototyping model

The prototype model can increase the cost of development for projects that are routine development work and do not suffer from any significant risks. The prototyping model would not be appropriate for projects for which the risks can only be identified after the development is underway.

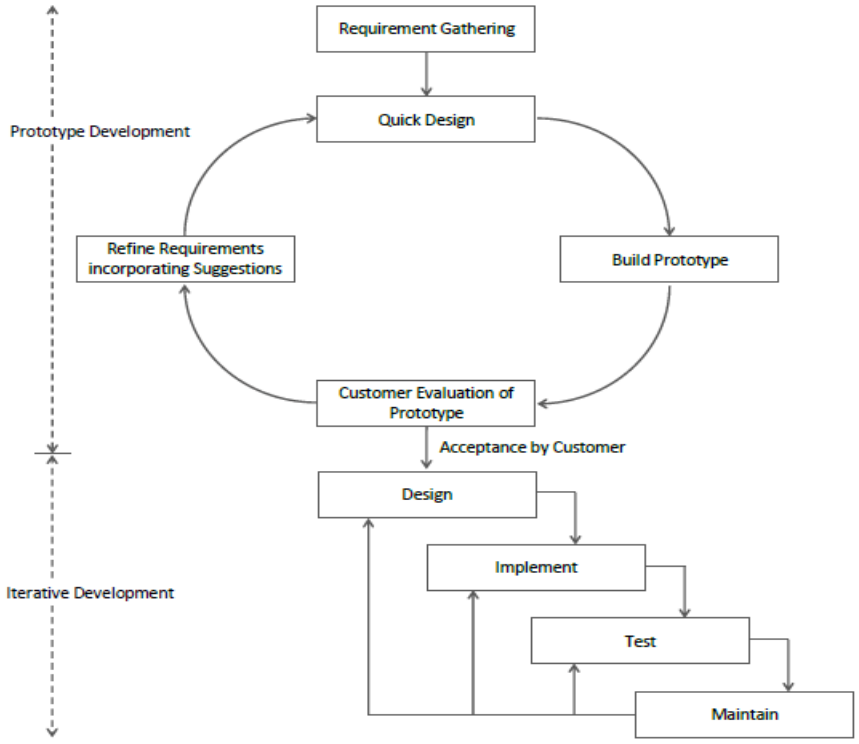


Figure: Prototyping model of software development

1.3.5. Incremental Development Model

This life cycle model is sometimes referred to as the successive versions model and sometimes as the incremental model. In this life cycle model, first a simple working system implementing only a few basic features is built and delivered to the customer. Over many successive iterations successive versions are implemented and delivered to the customer until the desired system is realised. The incremental development model has been shown in Figure 2.7.

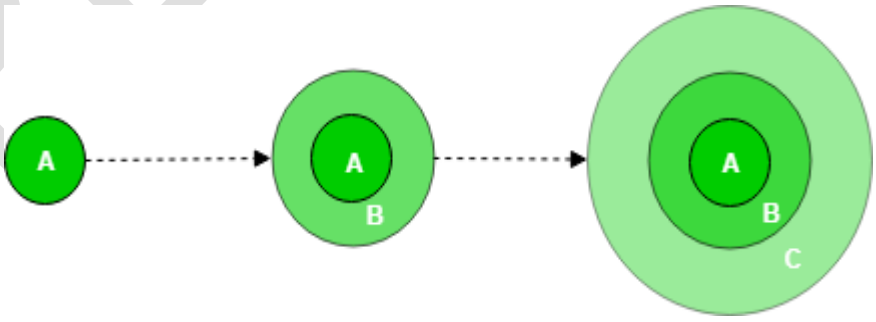


Figure: Incremental software development

Life cycle activities of incremental development model

In the incremental life cycle model, the requirements of the software are first broken down into several modules or features that can be incrementally constructed and delivered. This has been pictorially depicted in above figure. At any time, plan is made only for the next increment and no long-term plans are made. Therefore, it becomes easier to accommodate change requests from the customers.

The development team first undertakes to develop the **core features** of the system. The core or basic features are those that do not need to invoke any services from the other features. Once the initial core features are developed, these are refined into increasing levels of capability by adding new functionalities in successive versions. Each incremental version is usually developed using an iterative waterfall model of development. The incremental model is schematically shown in following figure. As each successive version of the software is constructed and delivered to the customer, the customer feedback is obtained on the delivered version and these feedbacks are incorporated in the next version. Each delivered version of the software incorporates additional features over the previous version and also refines the features that were already delivered to the customer.

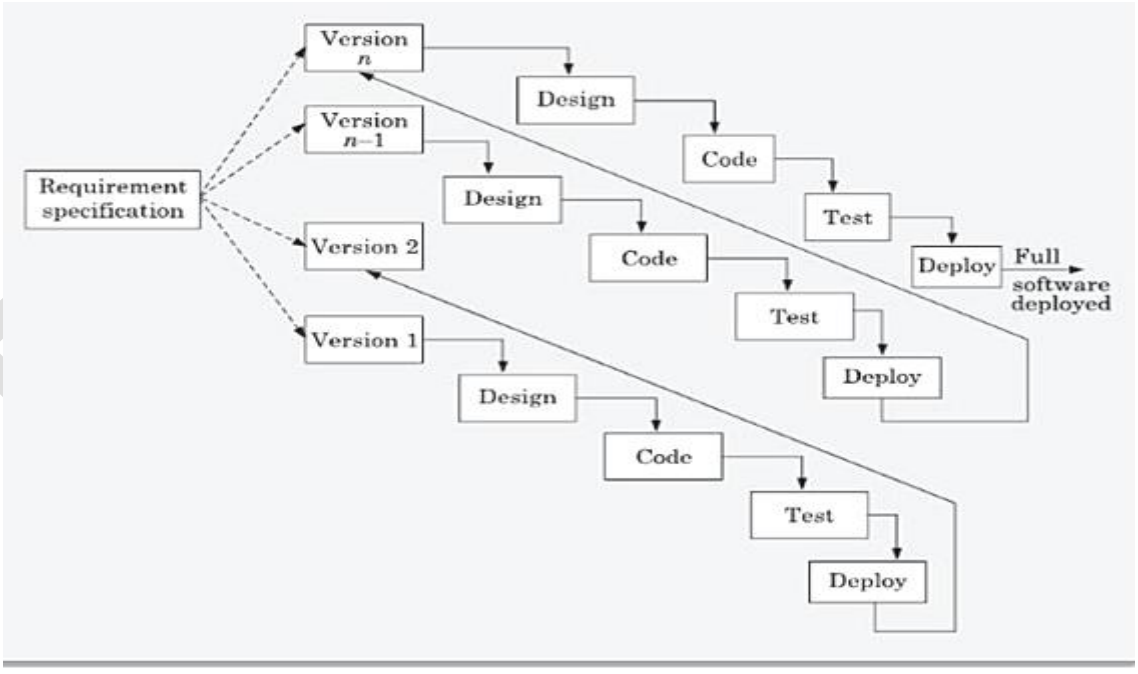


Figure : Incremental model of software development

Advantages

The incremental development model offers several advantages. Two important ones are the following:

Error reduction: The core modules are used by the customer from the beginning and therefore these get tested thoroughly. This reduces chances of errors in the core modules of the final product, leading to greater reliability of the software.

Incremental resource deployment: This model prevents the need for the customer to commit large resources at one go for development of the system. It also saves the developing organization from deploying large resources and manpower for a project in one go.

1.3.6. Evolutionary Model

This model has many of the features of the incremental model. As in case of the incremental model, the software is developed over a number of increments. At each increment, a concept (feature) is implemented and is deployed at the client site. The software is successively refined and feature-enriched until the full software is realized. The principal idea behind the evolutionary life cycle model is conveyed by its name. In the **incremental development model**, complete requirements are first developed and the SRS document prepared. In contrast, in the **evolutionary model**, the requirements, plan, estimates, and solution evolve over the iterations, rather than fully defined and frozen in a major up-front specification effort before the development iterations begin. Such evolution is consistent with the pattern of unpredictable feature discovery and feature changes that take place in new product development.

Though the evolutionary model can also be viewed as an extension of the waterfall model, but it incorporates a major paradigm shift that has been widely adopted in many recent life cycle models. The evolutionary software development process is sometimes referred to as ***design a little, build a little, test a little, deploy a little model.*** This means that after the requirements have been specified, the design, build, test, and deployment activities are iterated. A schematic representation of the evolutionary model of development has been shown in following figure.

Advantages

Two important advantages of using this model are the following:

Effective elicitation of actual customer requirements: In this model, the user gets a chance to experiment with a partially developed software much before the complete requirements are developed. Therefore, the evolutionary model helps to accurately elicit user requirements with the help of feedback obtained on the delivery of different versions of the software. As a result, the change requests after delivery of the complete software gets substantially reduced.

Easy handling change requests: In this model, handling change requests is easier as no long-term plans are made. Consequently, reworks required due to change requests are normally much smaller compared to the sequential models.

Disadvantages

The main disadvantages of the successive versions model are as follows:

Feature division into incremental parts can be non-trivial: For many development projects, especially for small-sized projects, it is difficult to divide the required features into several parts that can be incrementally implemented and delivered.

Ad hoc design: Since at a time design for only the current increment is done, the design can become ad hoc without specific attention being paid to maintainability and optimality. Obviously, for moderate sized problems and for those for which the customer requirements are clear, the iterative waterfall model can yield a better solution.

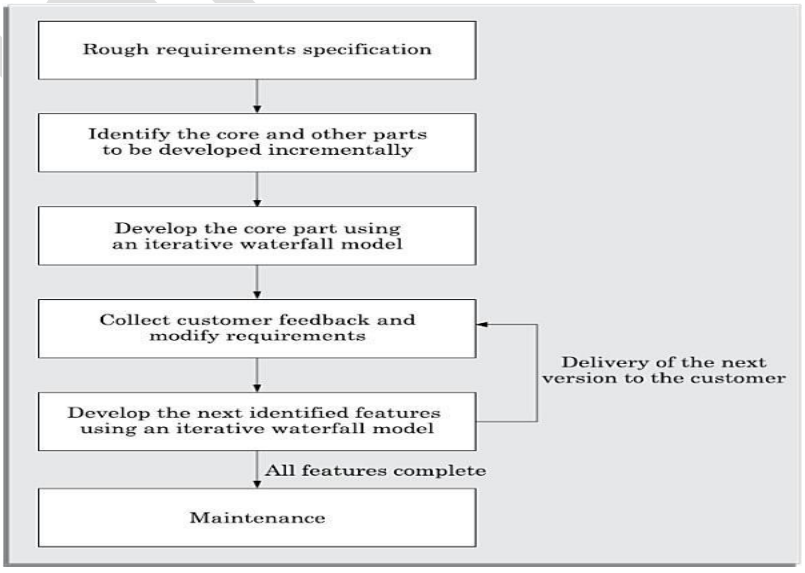


Figure: Evolutionary model of software development

The evolutionary model is normally useful for very large products, where it is easier to find modules for incremental implementation. Often evolutionary model is used when the customer prefers to receive the product in increments so that he can start using the different features as and when they are delivered rather than waiting all the time for the full product to be developed and delivered. The evolutionary model is well-suited to use in object-oriented software development projects.

1.3.7. RAPID APPLICATION DEVELOPMENT (RAD)

The *rapid application development* (RAD) model was proposed in the early nineties in an attempt to overcome the rigidity of the waterfall model that makes it difficult to accommodate any change requests from the customer. It proposed a few radical extensions to the waterfall model. This model has the **features of both prototyping and evolutionary models**. It deploys an evolutionary delivery model to obtain and incorporate the customer feedbacks on incrementally delivered versions.

In this model prototypes are constructed, and incrementally the features are developed and delivered to the customer. But unlike the prototyping model, the prototypes are not thrown away but are enhanced and used in the software construction

The major goals of the RAD model are as follows:

- To decrease the time taken and the cost incurred to develop software systems.
- To limit the costs of accommodating change requests.
- T o reduce the communication gap between the customer and the developers.

Working of RAD

In the RAD model, development takes place in a series of short cycles or iterations. At any time, the development team focuses on the present iteration only, and therefore plans are made for one increment at a time. The time planned for each iteration is called a ***time box*.** During each time box, **quick-and-dirty prototype-style** software for some functionality is developed. The customer evaluates the prototype and gives feedback on the specific improvements that may be necessary. The prototype is refined based on the customer feedback.

The development team almost always includes a customer representative to clarify the requirements. This is intended to make the system tuned to the exact customer requirements and also to bridge the communication gap between the customer and the development team.

The development team usually consists of about **five to six members**, including a customer representative.

The decrease in development time and cost, and at the same time an increased flexibility to incorporate changes are achieved in the RAD model in **two main ways**—**minimal use of planning** and **heavy reuse of any existing code through rapid prototyping**. **Reuse of existing code** has been adopted as an important mechanism of reducing the development cost.

RAD model emphasizes code reuse as an important means for completing a project faster. RAD advocates use of specialized tools to facilitate fast creation of working prototypes. These specialized tools usually support the following features:

- Visual style of development
- Use of reusable components

The following are some of the characteristics of an application that indicate its suitability to RAD style of development:

- **Customised software:** As already pointed out a customised software is developed for one or two customers only by adapting an existing software. In customised software development projects, substantial reuse is usually made of code from pre-existing software.
- **Non-critical software:** The RAD model suggests that a quick and dirty software should first be developed and later this should be refined into the final software for delivery.
- **Highly constrained project schedule:** RAD aims to reduce development time at the expense of good documentation, performance, and reliability. Naturally, for projects with very aggressive time schedules, RAD model should be preferred.
- **Large software:** Only for software supporting many features (large software) can incremental development and delivery be meaningfully carried out.

The RAD style of development is **not advisable** if a development project has one or more of the following characteristics

- **Generic products (wide distribution):** software products are generic in nature and usually have wide distribution. For such systems, optimal performance and reliability are imperative in a competitive market. The RAD model of development may not yield systems having optimal performance and reliability.
- **Requirement of optimal performance and/or reliability:** For certain categories of products, optimal performance or reliability is required. Examples of such systems include an

operating system (high reliability required) and a flight simulator software (high performance required). If such systems are to be developed using the RAD model, the desired product performance and reliability may not be realised.

- **Lack of similar products:** If a company has not developed similar software, then it would hardly be able to reuse much of the existing artifacts. In the absence of sufficient plug-in components, it becomes difficult to develop rapid prototypes through reuse, and use of RAD model becomes meaningless.
- **Monolithic entity:** For certain software, especially small-sized software, it may be hard to divide the required features into parts that can be incrementally developed and delivered. In this case, it becomes difficult to develop a software incrementally.

Comparison of RAD with Other Models

RAD *versus* prototyping model

In the prototyping model, the developed prototype is primarily used by the development team to gain insights into the problem, choose between alternatives, and elicit customer feedback. The code developed during prototype construction is usually thrown away. In contrast, in RAD it is the developed prototype that evolves into the deliverable software.

RAD *versus* iterative waterfall model

In the iterative waterfall model, all the functionalities of software are developed together. On the other hand, in the RAD model the product functionalities are developed incrementally through heavy code and design reuse. Further, in the RAD model customer feedback is obtained on the developed prototype after each iteration and based on this the prototype is refined. Thus, it becomes easy to accommodate any request for requirements changes. However, the iterative waterfall model does not support any mechanism to accommodate any requirement change requests. The iterative waterfall model does have some important advantages that include the following. Use of the iterative waterfall model leads to production of good quality documentation which can help during software maintenance. Also, the developed software usually has better quality and reliability than that developed using RAD.

RAD *versus* evolutionary model

Incremental development is the hallmark of both evolutionary and RAD models. However, in RAD each increment results in essentially a quick and dirty prototype, whereas in the evolutionary model each increment is systematically developed using the iterative waterfall model. Also in the RAD model, software is developed in much shorter increments compared the evolutionary model. In other words, the incremental functionalities that are developed are of fairly larger granularity in the evolutionary model.

1.3.8. SPIRAL MODEL

This model gets its name from the appearance of its diagrammatic representation that looks like a spiral with many loops. The exact number of loops of the spiral is not fixed and can vary from project to project. Each loop of the spiral is called a ***phase*** of the software process. The exact number of phases through which the product is developed can be varied by the project manager depending upon the project risks. A **prominent feature** of the spiral model is handling unforeseen risks that can show up much after the project has started.

In the spiral model prototypes are built at the start of every phase. Each phase of the model is represented as a loop in its diagrammatic representation. Over each loop, one or more features of the product are elaborated and analyzed and the risks at that point of time are identified and are resolved through prototyping. Based on this, the identified features are implemented.

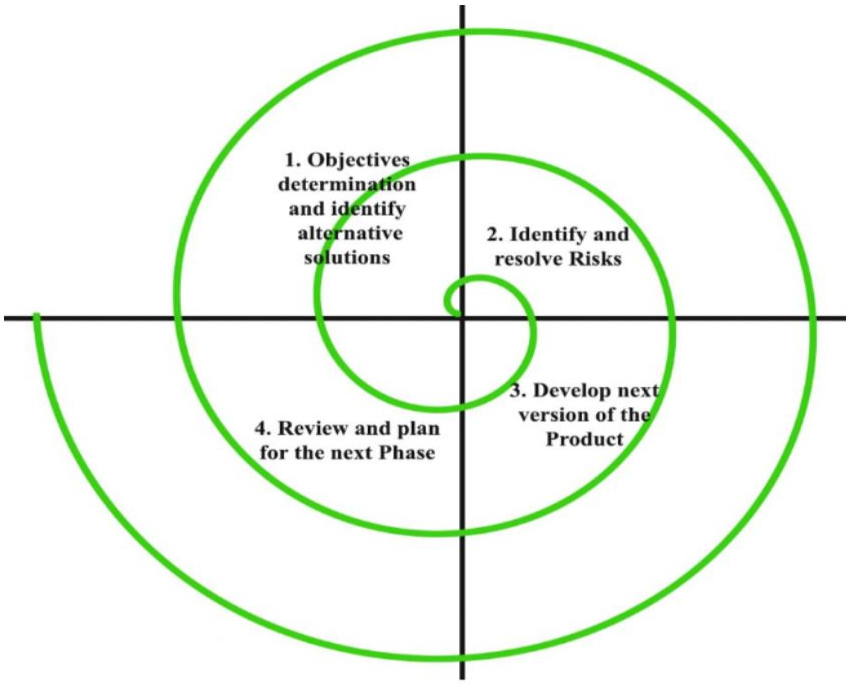


Figure: Spiral model of software development

Phases of the Spiral Model

Each phase in this model is split into **four sectors (or quadrants)** as shown in figure. In the **first quadrant**, a few features of the software are identified to be taken up for immediate development based on how crucial it is to the overall software development. With each iteration around the spiral (beginning at the center and moving outwards), progressively more complete versions of the software get built. In other words, implementation of the identified features forms a phase.

Quadrant 1: The objectives are investigated, elaborated, and analysed. Based on this, the risks involved in meeting the phase objectives are identified. In this quadrant, alternative solutions possible for the phase under consideration are proposed.

Quadrant 2: During the second quadrant, the alternative solutions are evaluated to select the best possible solution. To be able to do this, the solutions are evaluated by developing an appropriate prototype.

Quadrant 3: Activities during the third quadrant consist of developing and verifying the next level of the software. At the end of the third quadrant, the identified features have been implemented and the next version of the software is available.

Quadrant 4: Activities during the fourth quadrant concern reviewing the results of the stages traversed so far with the customer and planning the next iteration of the spiral.

The radius of the spiral at any point represents the cost incurred in the project so far, and the angular dimension represents the progress made so far in the current phase.

In the spiral model of development, the project manager dynamically determines the number of phases as the project progresses. Therefore, in this model, the project manager plays the crucial role of tuning the model to specific projects.

To make the model more efficient, the different features of the software that can be developed simultaneously through parallel cycles are identified.

Advantages/pros and disadvantages/cons of the spiral model

There are a few disadvantages of the spiral model that restrict its use to a only a few types of projects. To the developers of a project, the spiral model usually appears as a complex model to follow, since it is risk-driven and is more complicated phase structure than the other models we discussed. It would therefore be counterproductive to use, unless there are knowledgeable and experienced staff in the project. Also, it is not very suitable for use in the development of outsourced projects, since the software risks need to be continually assessed as it is developed.

Spiral model as a meta model

As compared to the previously discussed models, the spiral model can be viewed as a ***met a model***, since it subsumes all the discussed models. For example, a single loop spiral actually represents the waterfall model. The spiral model uses the approach of the prototyping model by first building a prototype in each phase before the actual development starts. This prototypes are used as a risk reduction mechanism. The spiral model incorporates the systematic step-wise approach of the waterfall model. Also, the spiral model can be considered as supporting the evolutionary model—the iterations along the spiral can be considered as evolutionary levels through which the complete system is built. This enables the developer to understand and resolve the risks at each evolutionary level.

Selecting an Appropriate Life Cycle Model for a Project

- **Characteristics of the software to be developed**
- **Characteristics of the development team**
- **Characteristics of the customer**

1.4.AGILE DEVELOPMENT MODELS

The agile software development model was proposed in the mid-1990s to overcome the serious shortcomings of the waterfall model.. The agile model was primarily designed to help a project to adapt to change requests **quickly**. Thus, a major aim of the agile models is to facilitate **quick project completion**.

Agile model is being used as an umbrella term to refer to a group of development processes. These processes share certain common characteristics, but do have certain subtle differences among themselves. A few popular **agile SDLC** models are the following:

- Crystal
- Atern (formerly DSDM)
- Feature-driven development
- Scrum
- Extreme programming (XP)
- Lean development
- Unified process

In the agile model, the requirements are decomposed into many small parts that can be incrementally developed. The agile model adopts an iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable and lasting for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer site. No long-term plans are made. The time to complete an iteration is called a ***time box***. The implication of the term *time box* is that the end date for an iteration does not change.

A central principle of the agile model is the delivery of an increment to the customer after each time box. Agile model emphasizes **face-to-face communication over written documents**. It is recommended that the development team size be deliberately kept small (5–9 people) to help the team members meaningfully engage in face-to-face communication and have collaborative work environment. It is implicit then that the agile model is suited to the development of small projects. However, if a large project is required to be developed using the agile model, it is likely that the collaborating teams might work at different locations. In this case, the different teams are needed to maintain as much daily contact as possible through video conferencing, telephone, e-mail, etc.

The following important principles behind the agile model were publicized in the agile manifesto in 2001:

- Working software over comprehensive documentation
- Frequent delivery of incremental versions of the software to the customer in intervals of few weeks.
- Requirement change requests from the customer are encouraged and are efficiently incorporated.
- Having competent team members and enhancing interactions among them is considered much more important than issues such as usage of sophisticated tools or strict adherence to a documented process.
- Continuous interaction with the customer is considered much more important rather than effective contract negotiation. A customer representative is required to be a part of the development team, thus facilitating close, daily co-operation between customers and developers.

Agile development projects usually deploy **pair programming**. In pair programming, **two** programmers work together at one work station. One types in code while the other reviews the code as it is typed in. The two programmers switch their roles every hour or so.

Advantages and disadvantages of agile methods

The agile methods derive much of their agility by relying on the tacit knowledge of the team members about the development project and informal communications to clarify issues, rather than spending significant amounts of time in preparing formal documents and reviewing them. Though this eliminates some overhead, but lack of adequate documentation may lead to several types of problems, which are as follows:

- Lack of formal documents leaves scope for confusion and important decisions taken during different phases can be misinterpreted at later points of time by different team members.
- In the absence of any formal documents, it becomes difficult to get important project decisions such as design decisions to be reviewed by external experts.
- When the project completes and the developers disperse, maintenance can become a problem.

Agile model ***versus*** iterative waterfall model

The **waterfall model** is highly structured and systematically steps through requirements-capture, analysis, specification, design, coding, and testing stages in a planned sequence. Progress is generally measured in terms of the number of completed and reviewed artifacts such as requirement specifications, design documents, test plans, code reviews, etc. for which review is complete. In contrast, while using an agile model, progress is measured in terms of the developed and delivered functionalities. In **agile model**, delivery of working versions of a software is made in several increments. However, as regards to similarity it can be said that agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration.

If a project being developed using **waterfall model** is cancelled mid-way during development, then there is nothing to show from the abandoned project beyond several documents. With **agile model**, even if a project is cancelled midway, it still leaves the customer with some worthwhile code, that might possibly have already been put into live operation.

Agile ***versus*** exploratory programming

Though a few similarities do exist between the **agile** and **exploratory programming** styles, there are vast differences between the two as well. **Agile development** model's frequent re-evaluation of plans, emphasis on face-to-face communication, and relatively sparse use of documentation are similar to that of the exploratory style. Agile teams, however, do follow defined and disciplined processes and carry out systematic requirements capture, rigorous designs, compared to chaotic coding in **exploratory programming**.

Agile model ***versus*** RAD model

The important differences between the agile and the RAD models are the following

- **Agile model** does not recommend developing prototypes, but emphasizes systematic development of each incremental feature. In contrast, the central theme of **RAD** is based on designing quick-and-dirty prototypes, which are then refined into production quality code.
- **Agile** projects logically break down the solution into features that are incrementally developed and delivered. The **RAD** approach does not recommend this. Instead, developers using the **RAD** model focus on developing all the features of an application by first doing it badly and then successively improving the code over time.

- **Agile** teams only demonstrate completed work to the customer. In contrast, **RAD** teams demonstrate to customers screen mock ups, and prototypes, that may be based on simplifications such as table look-ups rather than actual computations.

1.4.1. Extreme Programming (XP) Model

Extreme programming (XP) is an important process model under the agile umbrella and was proposed by **Kent Beck in 1999**. This model is based on a rather simple philosophy: ”If something is known to be beneficial, why not put it to constant use?” Based on this principle, it puts forward several key practices that need to be practiced to the extreme. Please note that most of the key practices that it emphasizes were already recognized as good practices for some time. In the following subsections, we mention some of the good practices that have been recognized in the extreme programming model and the suggested way to maximize their use:

Code review: It is good since it helps detect and correct problems most efficiently. It suggests ***pair programming*** as the way to achieve continuous review. In pair programming, coding is carried out by pairs of programmers. The programmers take turn in writing programs and while one writes the other reviews code that is being written.

Testing: Testing code helps to remove bugs and improves its reliability. XP suggests *test-driven development* (TDD) to continually write and execute test cases. In the TDD approach, test cases are written even before any code is written.

Incremental development: Incremental development is good, since it helps to get customer feedback, and extent of features delivered is a reliable indicator of progress. It suggests that the team should come up with new increments every few days.

Simplicity: Simplicity makes it easier to develop good quality code, as well as to test and debug it. Therefore, one should try to create the simplest code that makes the basic functionality being written to work. For creating the simplest code, one can ignore the aspects such as efficiency, reliability, maintainability, etc. Once the simplest thing works, other aspects can be introduced through refactoring.

Design: Since having a good quality design is important to develop a good quality solution, everybody should design daily. This can be achieved through ***refactoring***, whereby a working code is improved for efficiency and maintainability.

Integration testing: It is important since it helps identify the bugs at the interfaces of different functionalities. To this end, extreme programming suggests that the developers should achieve continuous integration, by building and performing integration testing several times a day.

XP is based on frequent releases (called *iteration*), during which the developers implement "**user stories**". User stories are similar to **use cases**, but are more informal and are simpler. A user story is the conversational description by the user about a feature of the required system.

On the basis of user stories, the project team proposes "**metaphors**"—a common vision of how the system would work. The development team may decide to construct a ***spike*** for some feature. A ***spike***, is a very simple program that is constructed to explore the suitability of a solution being proposed. A spike can be considered to be similar to a prototype.

X P prescribes several basic activities to be part of the software development process. These activities are:

Coding: XP argues that code is the crucial part of any system development process, since without code it is not possible to have a working system.

Testing: XP places high importance on testing and considers it be the primary means for developing a fault-free software.

Listening: The developers need to carefully listen to the customers if they have to develop a good quality software. Programmers may not necessarily be having an in-depth knowledge of the specific domain of the system under development. On the other hand, customers usually have this domain knowledge.

Designing: A good design should result in elimination of complex dependencies within a system. Thus, effective use of a suitable design technique is emphasized.

Feedback: It espouses the wisdom: "A system staying out of users is trouble waiting to happen". It recognizes the importance of user feedback in understanding the exact customer requirements.

Simplicity: A corner-stone of XP is based on the principle: "build something simple that will work today, rather than trying to build something that would take time and yet may never be used".

1.4.2. Scrum Model

In the scrum model, a project is divided into small parts of work that can be incrementally developed and delivered over time boxes that are called ***sprints***. Each sprint typically takes only a couple of weeks to complete. At the end of each sprint, stakeholders and team members meet to assess the progress made and the stakeholders suggest to the development team any changes needed to features that have already been developed and any overall improvements that they might feel necessary.

In the scrum model, the team members assume **three** fundamental roles— **software owner, scrum master, and team member**. The **software owner** is responsible for communicating the customers vision of the software to the development team. The **scrum master** acts as a liaison between the software owner and the team, thereby facilitating the development work.

1.5.SOFTWARE PROJECT MANAGEMENT

The main goal of software project management is to enable a group of developers to work effectively towards the successful completion of a project.

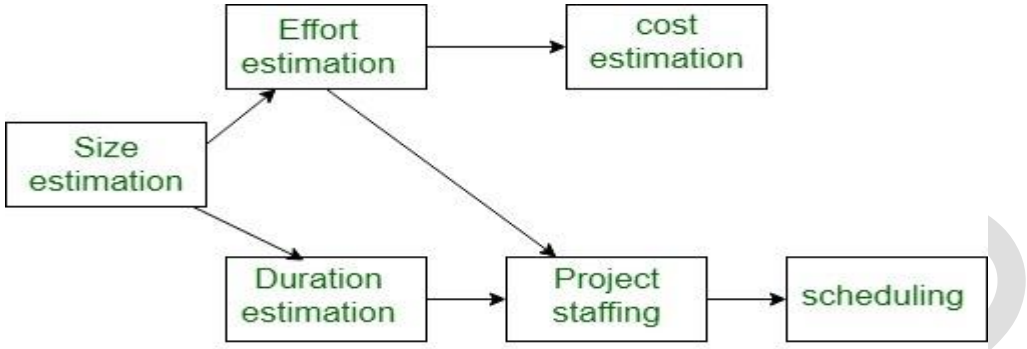
1.5.1.PROJECT PLANNING

Project planning is undertaken and completed before any development activity starts. Project planning requires utmost care and attention since commitment to unrealistic time and resource estimates result in schedule slippage. Schedule delays can cause customer dissatisfaction and adversely affect team morale. It can even cause project failure. For effective project planning, in addition to a thorough knowledge of the various estimation techniques, past experience is crucial. During project planning, the project manager performs the following activities.

- ✓ **Estimation:** The following project attributes are estimated.
  - **Cost:** How much is it going to cost to develop the software product?
  - **Duration:** How long is it going to take to develop the product?
  - **Effort:** How much effort would be necessary to develop the product?
- ✓ **Scheduling:** After all the necessary project parameters have been estimated, the schedules for manpower and other resources are developed.
- ✓ **Staffing:** Staff organization and staffing plans are made.
- ✓ **Risk management:** This includes risk identification, analysis, and abatement planning.

- ✓ **Miscellaneous plans:** This includes making several other plans such as quality assurance plan, and configuration management plan, etc.

Size is the most fundamental parameter based on which all other estimations and project plans are made.



Precedence ordering among planning activities

Sliding Window Planning

Planning a project over a number of stages protects managers from making big commitments at the start of the project. This technique of staggered planning is known as **sliding window planning**. i.e In the sliding window planning technique, starting with an initial plan, the project is planned more accurately over a number of stages.

The SPMP Document of Project Planning

Once project planning is complete, project managers document their plans in a **software project management plan (SPMP)** document. This list can be used as a possible organization of the **SPMP** document.

Organization of the software project management plan (SPMP) document

1. **Introduction**
   (a) Objectives
   (b) Major Functions
   (c) Performance Issues
   (d) Management and Technical Constraints
2. **Project estimates**
   (a) Historical Data Used
   (b) Estimation Techniques Used
   (c) Effort, Resource, Cost, and Project Duration Estimates
3. **Schedule**
   (a) Work Breakdown Structure
   (b) Task Network Representation
   (c) Gantt Chart Representation

(d) PERT Chart Representation

4. **Project resources**
   (a) People
   (b) Hardware and Software
   (c) Special Resources
5. **Staff organization**
   (a) Team Structure
   (b) Management Reporting
6. **Risk management plan**
   (a) Risk Analysis
   (b) Risk Identification
   (c) Risk Estimation
   (d) Risk Abatement Procedures
7. **Project tracking and control plan**
   (a) Metrics to be tracked
   (b) Tracking plan
   (c) Control plan

8. **Miscellaneous plans**
   (a) Process Tailoring
   (b) Quality Assurance Plan
   (c) Configuration Management Plan
   (d) Validation and Verification
   (e) System Testing Plan
   (f) Delivery, Installation, and Maintenance Plan

1.5.2. METRICS FOR PROJECT SIZE ESTIMATION

The size of a project is obviously not the number of bytes that the source code occupies, neither is it the size of the executable code. The project size is a measure of the problem complexity in terms of the effort and time required to develop the product. There are **two** metrics are popularly being used to measure size—**lines of code (LOC)** and **function point (FP)**.

Lines of Code (LOC)

LOC is possibly the simplest among all metrics available to measure project size. This metric measures the size of a project by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, comment lines, and header lines are ignored.

Function Point (FP) Metric

Function point metric was proposed by Albrecht in 1983. Function point metric has several advantages over LOC metric. One of the important advantages of the function point metric over the LOC metric is that it can easily be computed from the problem specification itself. Using the LOC metric, on the other hand, the size can accurately be determined only after the product has fully been developed.

Function point (FP) metric computation

The size of a software product (in units of function points or FPs) is computed using different characteristics of the product identified in its requirements specification. It is computed using the following three steps:

Step 1: Compute the **unadjusted function point (UFP)** using a heuristic expression.

Step 2: Refine UFP to reflect the actual complexities of the different parameters used in UFP computation.

Step 3: Compute FP by further refining UFP to account for the specific characteristics of the project that can influence the entire development effort.

Step 1: UFP computation

The unadjusted function points (UFP) is computed as the weighted sum of **five** characteristics

The weights associated with the **five** characteristics were determined empirically by Albrecht through data gathered from many projects.

UFP = (Number of inputs)*4 + (Number of outputs)*5 + (Number of inquiries)*4 + (Number of files)*10 + (Number of interfaces)*10

Step 2: Refine parameters

The complexity of each parameter is graded into three broad categories—simple, average, or complex. The weights for the different parameters are determined based on the numerical values shown in following table. Based on these weights of the parameters, the parameter values in the UFP are refined.

Type	Simple	Average	Complex
Input(I)	3	4	6
Output (O)	4	5	7
Inquiry (E)	3	4	6
Number of files (F)	7	10	15

Number of interfaces	5	7	10

Table: Refinement of Function Point Entities

Step 3: Refine UFP based on complexity of the overall project

Albrecht identified 14 parameters that can influence the development effort. The list of these parameters has been shown in following list. Each of these 14 parameters is assigned a value from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). A technical complexity factor (TCF) for the project is computed and the TCF is multiplied with UFP to yield FP. The TCF expresses the overall impact of the corresponding project parameters on the development effort.

TCF is computed as (0.65+0.01*DI).

As DI can vary from 0 to 84, TCF can vary from 0.65 to 1.49. Finally, FP is given as the product of UFP and TCF.

That is, **FP=UFP*TCF.**

Function Point Relative Complexity Adjustment Factors are :

1. Requirement for reliable backup and recovery
2. Requirement for data communication
3. Extent of distributed processing
4. Performance requirements
5. Expected operational environment
6. Extent of online data entries
7. Extent of multi-screen or multi-operation online data input
8. Extent of online updating of master files
9. Extent of complex inputs, outputs, online queries and files
10. Extent of complex data processing
11. Extent that currently developed code can be designed for reuse
12. Extent of conversion and installation included in the design
13. Extent of multiple installations in an organization and variety of customer organizations
14. Extent of change and focus on ease of use

1.5.3. PROJECT ESTIMATION TECHNIQUES

Estimation of various project parameters is an important project planning activity. The different parameters of a project that need to be estimated include—**project size, effort required to**

complete the project, project duration, and cost. A large number of estimation techniques have been proposed by researchers. These can broadly be classified into **three** main categories:

1. **Empirical estimation techniques**
2. **Heuristic techniques**
3. **Analytical estimation techniques**

1.5.3.1.Empirical Estimation Techniques

Empirical estimation techniques are essentially based on making an educated guess of the project parameters. While using this technique, prior experience with development of similar products is helpful. Although empirical estimation techniques are based on common sense and subjective decisions, over the years, the different activities involved in estimation have been formalized to a large extent. Two popular empirical estimation techniques are **expert judgement** and **the Delphi techniques.**

Expert Judgement

Expert judgement is a widely used size estimation technique. In this technique, an expert makes an educated guess about the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) that would make up the system and then combines the estimates for the individual modules to arrive at the overall estimate. However, this technique suffers from several shortcomings. The outcome of the expert judgement technique is subject to human errors and individual bias. Also, it is possible that an expert may overlook some factors inadvertently. Further, an expert making an estimate may not have relevant experience and knowledge of all aspects of a project.

A more refined form of expert judgement is the estimation made by a group of experts. Chances of errors arising out of issues such as individual oversight, lack of familiarity with a particular aspect of a project, personal bias, and the desire to win contract through overly optimistic estimates is minimized when the estimation is done by a group of experts. However, the estimate made by a group of experts may still exhibit bias.

Delphi Cost Estimation

Delphi cost estimation technique tries to overcome some of the shortcomings of the expert judgement approach. Delphi estimation is carried out by a team comprising a group of experts and a coordinator. In this approach, the coordinator provides each estimator with a copy of the **software requirements specification (SRS)** document and a form for recording his cost estimate. Estimators complete their individual estimates anonymously and submit them to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influence d their estimations. The coordinator prepares the summary of the responses of all the estimators, and also includes any unusual rationale noted by any of the estimators. The prepared summary information is distributed to the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussions among the estimators is allowed during the entire estimation process. The purpose behind this restriction is that if any discussion is allowed among the estimators, then many estimators may easily get influenced by the rationale of an estimator who may be more experienced or senior. After the completion of several iterations of estimations, the coordinator takes the responsibility of compiling the results and preparing the final estimate.

1. 5.3.2. COCOMO—A HEURISTIC ESTIMATION TECHNIQUE

COnstructive COst estimation MOdel (COCOMO) was proposed by Boehm in 1981. **COCOMO** prescribes a **three-stage** process for project estimation. In the **first stage**, an initial estimate is arrived at. Over the **next two stages**, the initial estimate is refined to arrive at a more accurate estimate. COCOMO uses both single and multivariable estimation models at different stages of estimation.

The **three** stages of COCOMO estimation technique are—**basic COCOMO, intermediate COCOMO,** and **complete COCOMO**.

Basic COCOMO Model

Boehm postulated that any software development project can be classified into one of the following **three** categories based on the development complexity—**organic, semidetached, and embedded**. Based on the category of a software development project, he gave different sets of formulas to estimate the effort and duration from the size estimate.

Boehm's definitions of **organic, semidetached,** and **embedded** software are elaborated as follows:

- **Organic:** We can classify a development project to be of organic type, if the project deals with developing a well-understood application program, the size of the development team is reasonably small, and the team members are experienced in developing similar types of projects.
- **Semidetached:** A development project can be classify to be of semidetached type, if the development team consists of a mixture of experienced and inexperienced staff. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.
- **Embedded:** A development project is considered to be of embedded type, if the software being developed is strongly coupled to hardware, or if stringent regulations on the operational procedures exist. Team members may have limited experience on related systems but may be unfamiliar with some aspects of the system being developed.

The **basic COCOMO model** is a single variable heuristic model that gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by expressions of the following forms:

- $Effort = a1 \times (KLOC)^{a2}$ PM
- $Tdev = b1 \times (Effort)^{b2}$ months

were,

- KLOC is the estimated size of the software product expressed in Kilo Lines Of Code
- a_1, a_2, b_1, b_2 are constants for each category of software product
- Tdev is the estimated time to develop the software, expressed in months
- Effort is the total effort required to develop the software product, expressed in person-months (PMs).

Estimation of development effort: For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic : Effort = 2.4(KLOC)1.05 PM

Semi-detached : Effort = 3.0(KLOC)1.12 PM

Embedded : Effort = 3.6(KLOC)1.20 PM

Estimation of development time: For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic : Tdev = 2.5(Effort)0.38 Months

Semi-detached : Tdev = 2.5(Effort)0.35 Months

Embedded : Tdev = 2.5(Effort)0.32 Months

Intermediate COCOMO

The intermediate COCOMO model refines the initial estimate obtained using the basic COCOMO expressions by scaling the estimate up or down based on the evaluation of a set of attributes of software development

The intermediate COCOMO model uses a set of 15 cost drivers that are determined based on various attributes of software development. These cost drivers are multiplied with the initial cost and effort estimates (obtained from the basic COCOMO) to appropriately scale those up or down.

The cost drivers identified by Boehm can be classified as being attributes of the following items:

Product: The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.

Computer: Characteristics of the computer that are considered include the execution speed required, storage space required, etc.

Personnel: The attributes of development personnel that are considered include the experience level of personnel, their programming capability, analysis capability, etc.

Development environment: Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development.

Complete COCOMO

The complete COCOMO model considers these differences in characteristics of the subsystems and estimates the effort and development time as the sum of the estimates for the individual sub-systems.

In other words, the cost to develop each sub-system is estimated separately, and the complete system cost is determined as the subsystem costs. This approach reduces the margin of error in the final estimate.

1.5.3.3. HALSTEAD'S SOFTWARE SCIENCE

Halstead's software science is an analytical technique to measure size, development effort, and development cost of software products. Halstead used a few primitive program parameters to develop the expressions for overall program length, potential minimum volume, actual volume, language level, effort, and development time. For a given program, let:

- h_1 be the number of unique operators used in the program,
- h_2 be the number of unique operands used in the program,
- N_1 be the total number of operators used in the program,
- N_2 be the total number of operands used in the program.

1.5.4. PROJECT SCHEDULING

Scheduling the project tasks is an important project planning activity. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the major activities that need to be carried out to complete the project.
2. Break down each activity into tasks.
3. Determine the dependency among different tasks.
4. Establish the estimates for the time durations necessary to complete the tasks.
5. Represent the information in the form of an activity network.
6. Determine task starting and ending dates from the information represented in the activity network.
7. Determine the critical path. A critical path is a chain of tasks that determines the duration of the project.

8. Allocate resources to tasks

The first step in scheduling a software project involves identifying all the activities necessary to complete the project. A good knowledge of the intricacies of the project and the development process helps the managers to effectively identify the important activities of the project. Next, the activities are broken down into a logical set of smaller activities (sub activities). The smallest sub activities are called **tasks** which are assigned to different developers. The smallest units of work activities that are subject to management planning and control are called **tasks**.

Project manager breakdowns the tasks systematically by using the **work breakdown structure** technique. After the project manager has broken down the activities into tasks, he has to find the dependency among the tasks. Dependency among the different tasks determines the order in which the different tasks would be carried out.

Work Breakdown Structure

Work breakdown structure (WBS) is used to recursively decompose a given set of activities into smaller activities. First, let us understand why it is necessary to break down project activities into tasks. Once project activities have been decomposed into a set of tasks using WBS, the time frame when each activity is to be performed is to be determined. The end of each important activity is called a **milestone**. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that some milestones start getting delayed, he carefully monitors and controls the progress of the tasks, so that the overall deadline can still be met.

WBS provides a notation for representing the activities, sub-activities, and tasks needed to be carried out in order to solve a problem. Each of these is represented using a rectangle. The root of the tree is labeled by the project name. Each node of the tree is broken down into smaller activities that are made the children of the node. To decompose an activity to a sub-activity, a good knowledge of the activity can be useful. The following figure represents the WBS of a management information system (MIS) software.

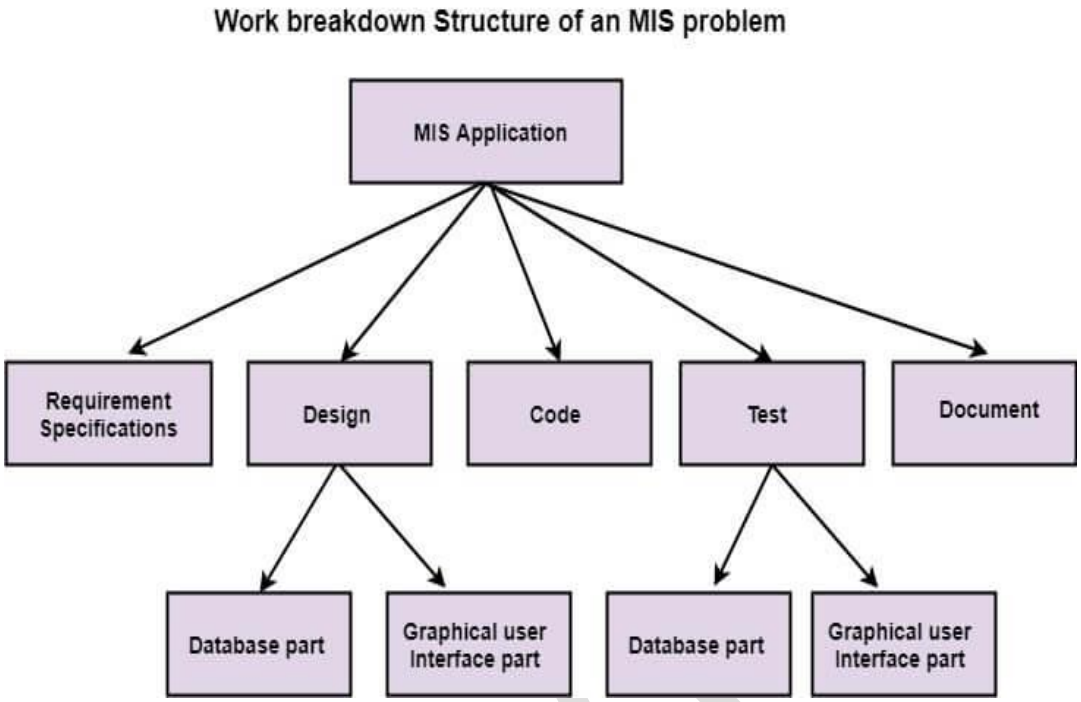
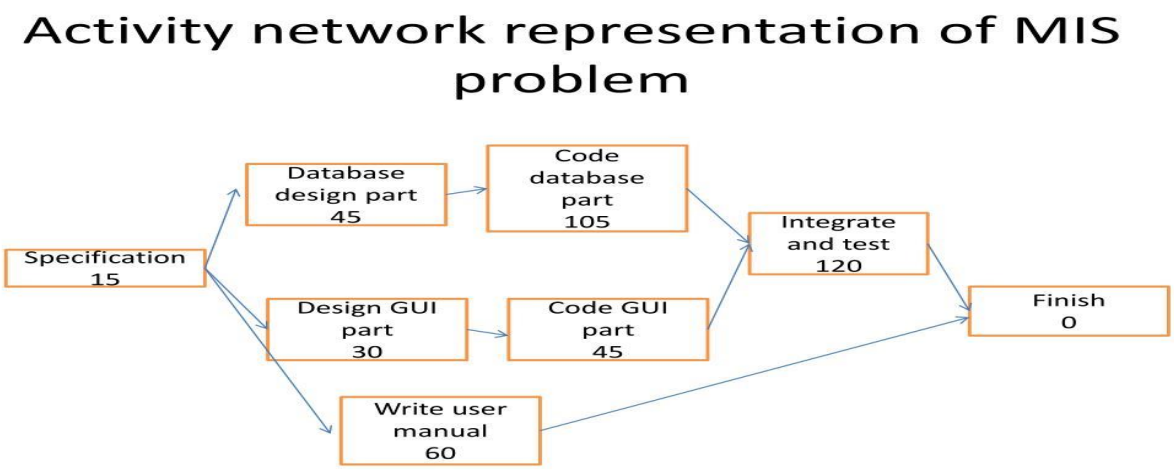


Figure: Work breakdown structure of an MIS problem

Activity Networks

An activity network shows the different activities making up a project, their estimated durations, and their interdependencies. **Two** equivalent representations for activity networks are possible and are in use:

Activity on Node (AoN): In this representation, each activity is represented by a rectangular (some use circular) node and the duration of the activity is shown alongside each task in the node. The inter-task dependencies are shown using directional edges.



Activity on Edge (AoE): In this representation tasks are associated with the edges. The edges are also annotated with the task duration. The nodes in the graph represent project milestones.

Critical Path Method (CPM)

CPM and PERT are operation research techniques that were developed in the late 1950s. Since then, they have remained extremely popular among project managers.

A path in the activity network graph is any set of consecutive **nodes** and **edges** in this graph from the starting node to the last node. A critical path consists of a set of dependent tasks that need to be performed in a sequence and which together take the longest time to complete.

CPM is an algorithmic approach to determine the critical paths and slack times for tasks not on the critical paths involves calculating the following quantities:

Minimum time (MT): It is the minimum time required to complete the project. It is computed by determining the maximum of all paths from start to finish.

Earliest start (ES): It is the time of a task is the maximum of all paths from the start to this task. The ES for a task is the ES of the previous task plus the duration of the preceding task.

Latest start time (LST): It is the difference between MT and the maximum of all paths from this task to the finish. The LST can be computed by subtracting the duration of the subsequent task from the LST of the subsequent task.

Earliest finish time (EF): The EF for a task is the sum of the earliest start time of the task and the duration of the task.

Latest finish (LF): LF indicates the latest time by which a task can finish without affecting the final completion time of the project. A task completing beyond its LF would cause project delay. LF of a task can be obtained by subtracting maximum of all paths from this task to finish from MT.

Slack time (ST): The slack time (or float time) is the total time that a task may be delayed before it will affect the end time of the project. The slack time indicates the ”flexibility” in starting and completion of tasks. ST for a task is LS-ES and can equivalently be written as LF-EF.

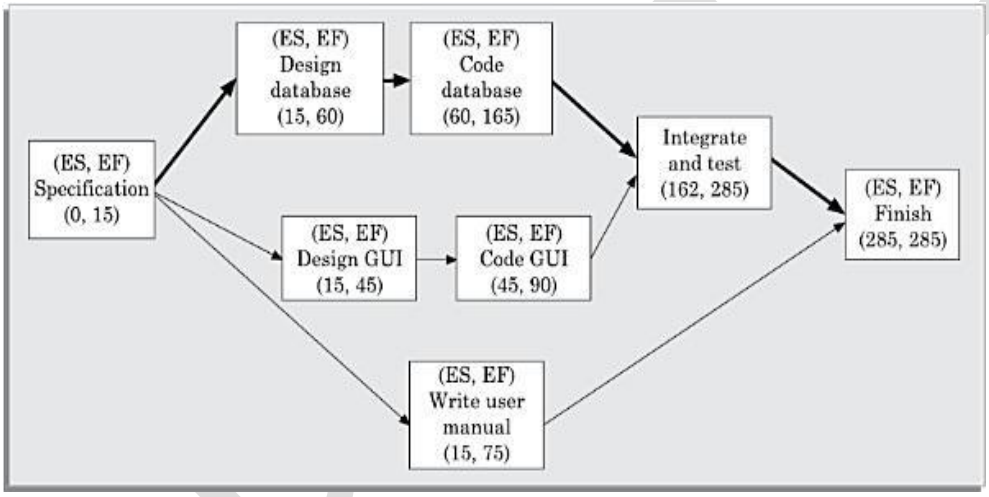


Figure: AoN for MIS problem with (ES,EF).

PERT Charts

The activity durations computed using an activity network are only estimated duration. It is therefore not possible to estimate the worst case (pessimistic) and best case (optimistic) estimations using an activity diagram. Since, the actual durations might vary from the estimated durations, the utility of the activity network diagrams are limited. The CPM can be used to determine the duration of a project, but does not provide any indication of the probability of meeting that schedule.

Project evaluation and review technique (PERT) charts are a more sophisticated form of activity chart. Project managers know that there is considerable uncertainty about how much time a task would exactly take to complete. The duration assigned to tasks by the project manager are after all only estimates. In this context, PERT charts can be used to determine the

probabilistic times for reaching various project mile stones, including the final mile stone. PERT charts like activity networks consist of a network of boxes and arrows. The **boxes** represent **activities** and the **arrows** represent **task** dependencies. A PERT chart represents the statistical variations in the project estimates assuming these to be normal distribution. PERT allows for some randomness in task completion times, and therefore provides the capability to determine the probability for achieving project milestones based on the probability of completing each task along the path to that milestone. Each task is annotated with **three** estimates:

- **Optimistic (O):** The best possible case task completion time.
- **Most likely estimate (M):** Most likely task completion time
- **Worst case (W):** The worst possible case task completion time

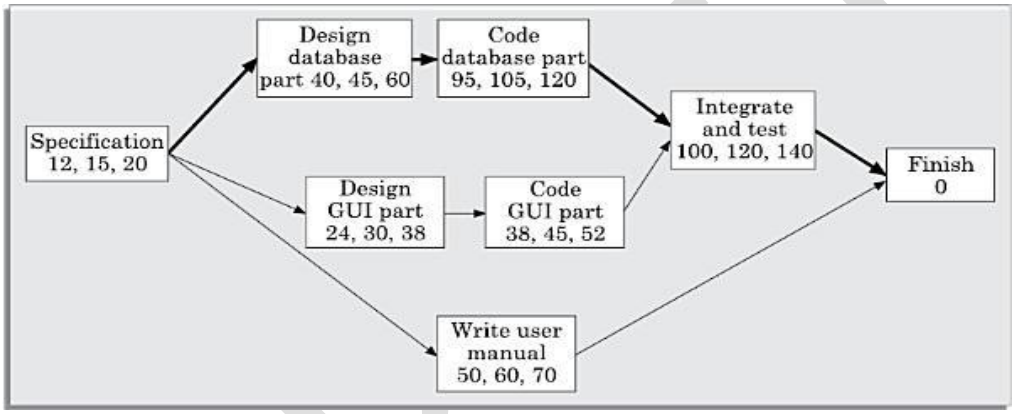


Figure: PERT chart representation of the MIS problem

Gantt Charts

Gantt chart has been named after its developer Henry Gantt. A Gantt chart is a form of bar chart. The vertical axis lists all the tasks to be performed. The bars are drawn along the y-axis, one for each task. Gantt charts used in software project management are actually an enhanced version of the standard Gantt charts. In the Gantt charts used for software project management, each bar consists of a unshaded part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The unshaded part shows the slack time or lax time. The lax time represents the leeway or flexibility available in meeting the latest time by which a task must be finished. A Gantt chart representation for the MIS problem shown in following figure. Gantt charts are useful for resource planning (i.e. allocate resources to activities). The different types of resources that need to be allocated to activities include staff, hardware, and software.

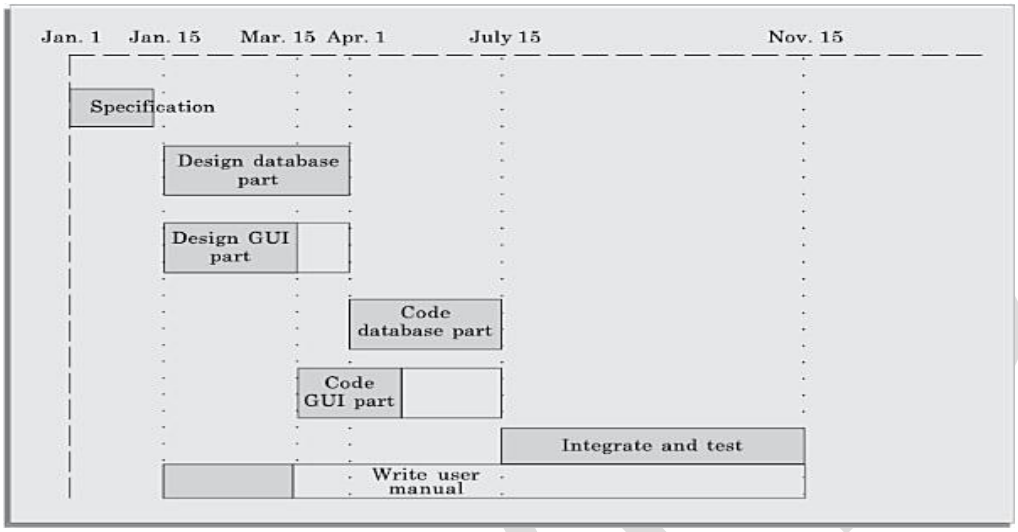


Figure: Gantt chart representation of the MIS problem.

Gantt chart representation of a project schedule is helpful in planning the utilization of resources, while PERT chart is useful for monitoring the timely progress of activities. Also, it is easier to identify parallel activities in a project using a PERT chart. Project managers need to identify the parallel activities in a project for assignment to different developers.

1.6. ORGANISATION AND TEAM STRUCTURES

Usually every software development organization handles several projects at any time. Software organizations assign different teams of developers to handle different software projects. With regard to staff organization, there are **two** important issues—**How is the organization as a whole structured?** And, **how are the individual project teams structured**?

Organisation Structure

Essentially there are **three** broad ways in which a software development organisation can be structured—**functional format, project format, and matrix format**.

Functional format

In the functional format, the development staff are divided based on the specific functional group to which they belong to. This format has schematically been shown in Figure. The different

projects borrow developers from various functional groups for specific phases of the project and return them to the functional group upon the completion of the phase.

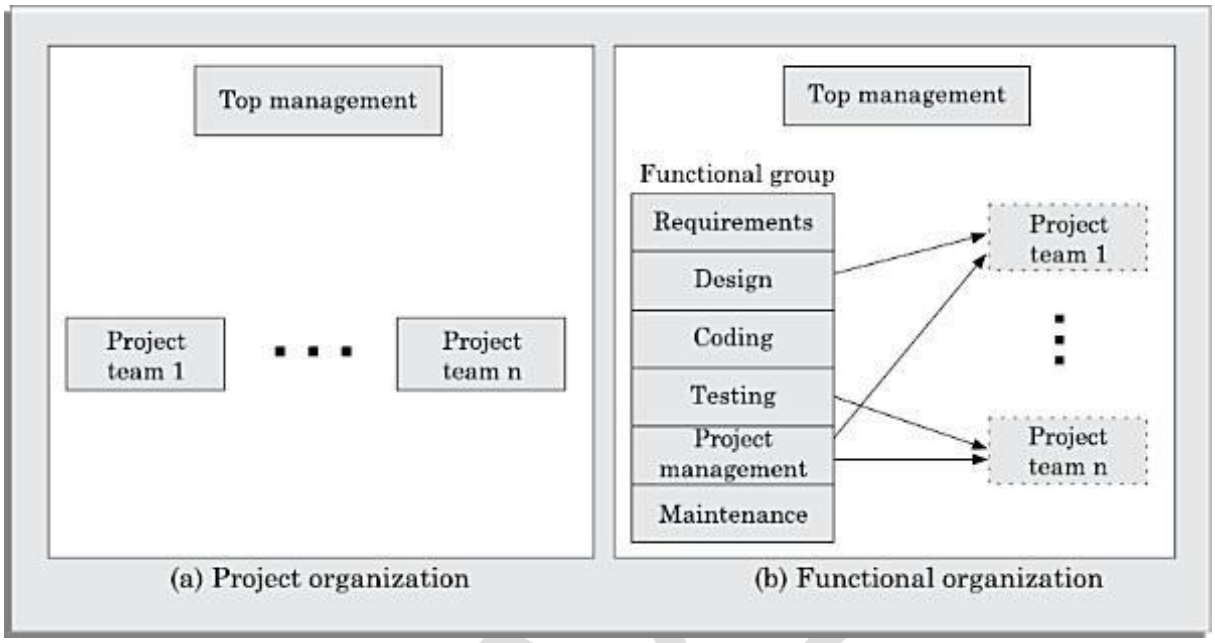


Figure: Schematic representation of the functional and project organisation

Project format

In the project format, the development staff are divided based on the project for which they work. A set of developers is assigned to every project at the start of the project, and remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. An advantage of the project format is that it provides job rotation. That is, every developer undertakes different life cycle activities in a project. However, it results in poor manpower utilisation, since the full project team is formed since the start of the project, and there is very little work for the team during the initial phases of the life cycle.

Functional versus project formats

Even though greater communication among the team members may appear as an avoidable overhead, the functional format has many advantages. The main advantages of a functional organisation are :

- Ease of staffing
- Production of good quality documents
- Job specialization
- Efficient handling of the problems associated with manpower turnover

The functional organisation allows the developers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. The functional organisation also provides an efficient solution to the staffing problem. The project staffing problem is eased significantly because personnel can be brought onto a project as needed, and returned to the functional group when they are no more needed. This possibly is the most important advantage of the functional organisation.

A project organisation structure forces the manager to take in almost a constant number of developers for the entire duration of his project. This results in developers idling in the initial phase of software development and are under tremendous pressure in the later phase of development.

Matrix format

A matrix organisation is intended to provide the advantages of both functional and project structures. In a matrix organisation, the pool of functional specialists are assigned to different projects as needed. Thus, the deployment of the different functional specialists in different projects can be represented in a matrix. In following figure observe that different members of a functional specialisation are assigned to different projects. Therefore in a matrix organisation, the project manager needs to share responsibilities for the project with a number of individual functional managers.

	Project			
Functional group	#1	#2	#3	
#1	2	0	3	Functional manager 1
#2	0	5	3	Functional manager 2
#3	0	4	2	Functional manager 3
#4	1	4	0	Functional manager 4
#5	0	4	6	Functional manager 5
	Project manager 1	Project manager 2	Project manager 3	

Figure : Matrix organisation

Matrix organisations can be characterised as **weak or strong**, depending upon the relative authority of the functional managers and the project managers. In a strong functional matrix, the

functional managers have authority to assign workers to projects and project managers have to accept the assigned personnel. In a weak matrix, the project manager controls the project budget, can reject workers from functional groups, or even decide to hire outside workers.

Two important problems that a matrix organisation often suffers from are:

- Conflict between functional manager and project managers over allocation of workers.
- Frequent shifting of workers in a firefighting mode as crises occur in different projects

Team Structure

Team structure addresses organisation of the individual project teams. There are **three** formal team structures—**democratic, chief programmer, and the mixed control team organisations**.

Democratic team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy. Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership. In a democratic organisation, the team members have higher morale and job satisfaction. Consequently, it suffers from less manpower turnover. The democratic team organisation encourages egoless programming as programmers can share and review each other's work.

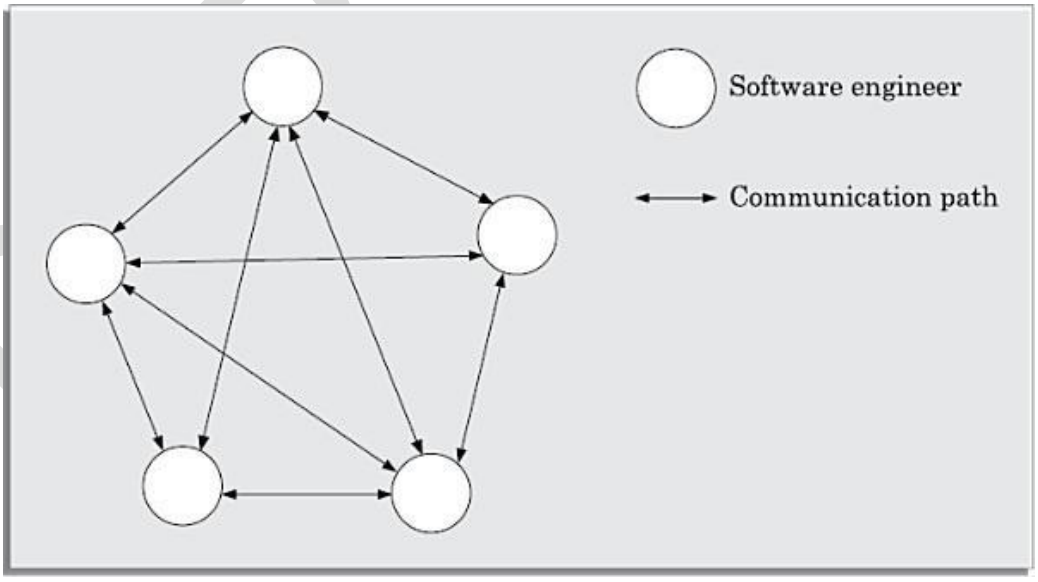


Figure: Democratic team structure

Chief programmer team

In this team organisation, a senior engineer provides the technical leadership and is designated the chief programmer. The chief programmer partitions the task into many smaller tasks and assigns them to the team members. He also verifies and integrates the products developed by different team members. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. The chief programmer team is subject to **single point failure** since too much responsibility and authority is assigned to the chief programmer. That is, a project might suffer severely, if the chief programmer either leaves the organisation or becomes unavailable for some other reasons.

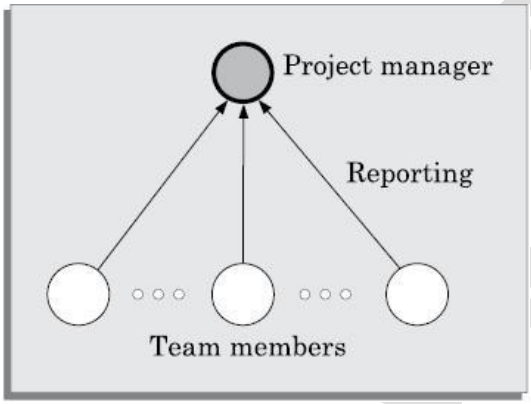


Figure: Chief programmer team structure

Mixed control team organisation

The mixed control team organisation, as the name implies, draws upon the ideas from both the **democratic organisation** and the **chief-programmer organisation**. This team organisation incorporates both hierarchical reporting and democratic set up. The communication paths are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organisation is suitable for large team sizes.. This team structure is extremely popular and is being used in many software development companies.

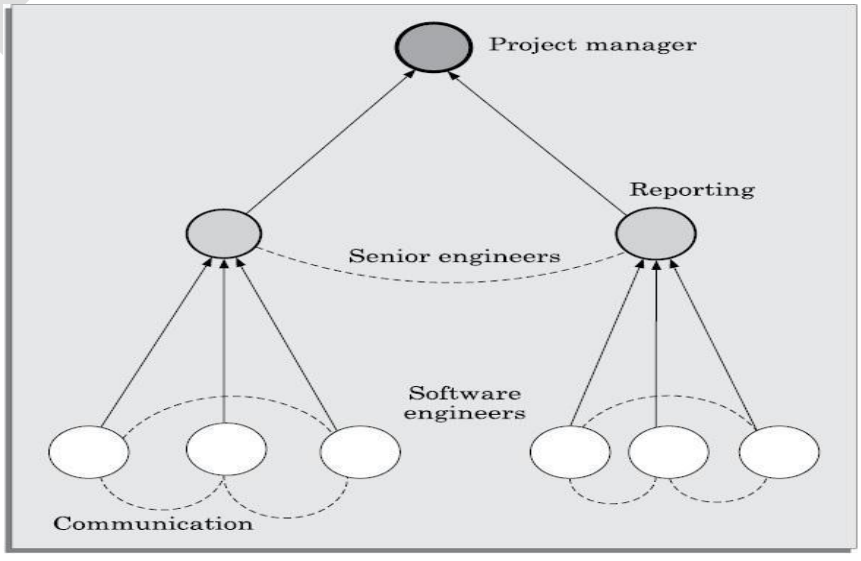


Figure: Mixed team structure

1.7. STAFFING

Software project managers usually take the responsibility of choosing their team. They need to identify good software developers for the success of the project.

Who is a good software engineer?

The following attributes that good software developers should possess:

- Exposure to systematic techniques, i.e. familiarity with software engineering principles
- Good technical knowledge of the project areas (Domain knowledge)
- Good programming abilities.
- Good communication skills. These skills comprise of oral, written, and interpersonal skills.
- High motivation
- Sound knowledge of fundamentals of computer science
- Intelligence.
- Ability to work in a team
- Discipline, etc

Technical knowledge in the area of the project (domain knowledge) is an important factor determining the productivity of an individual for a particular project, and the quality of the product that he develops. Since software development is a group activity, it is vital for a software developer to possess **three** main kinds of communication skills—**Oral, Written, and Interpersonal**. Poor interpersonal skills hamper these vital activities and often show up as poor quality of the product and low productivity. Software developers are also required at times to make presentations to the managers and to the customers.

Motivation level of a software developer is another crucial factor contributing to his work quality and productivity. Even though no systematic studies have been reported in this regard, it is generally agreed that even bright developers may turn out to be poor performers when they lack motivation. An average developer who can work with a single mind track can outperform other developers. But motivation is a complex phenomenon requiring careful control. For majority of software developers, higher incentives and better working conditions have only limited affect on their motivation levels. Motivation is to a great extent determined by personal traits, family and social backgrounds, etc.

1.8. RISK MANAGEMENT

Every project is susceptible to a large number of risks. A risk is any anticipated unfavorable event or circumstance that can occur while a project is underway. Risk management aims at reducing the chances of a risk becoming real as well as reducing the impact of a risks that becomes real. Risk management consists of **three** essential activities—**risk identification, risk assessment, and risk mitigation**.

Risk Identification

The project manager needs to anticipate the risks in a project as early as possible. As soon as a risk is identified, effective risk management plans are made, so that the possible impacts of the risks are minimized. So, early risk identification is important. Risk identification is somewhat similar to the project manager listing down his nightmares.

A project can be subject to a large variety of risks. In order to be able to systematically identify the important risks which might affect a project, it is necessary to categorize risks into different classes. There are **three** main categories of risks which can affect a software project: **project risks, technical risks**, and **business risks**.

Project risks: Project risks concern various forms of budgetary, schedule, personnel, resource, and customer-related problems. An important project risk is schedule slippage. Since, software is intangible, it is very difficult to monitor and control a software project. It is very difficult to control something which cannot be seen.

Technical risks: Technical risks concern potential design, implementation, interfacing, testing, and maintenance problems. Technical risks also include ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks occur due the development team's insufficient knowledge about the product.

Business risks: This type of risks includes the risk of building an excellent product that no one wants, losing budgetary commitments, etc.

Risk Assessment

The objective of risk assessment is to rank the risks in terms of their damage causing potential. For risk assessment, first each risk should be rated in **two** ways:

- The likelihood of a risk becoming real (r).
- The consequence of the problems associated with that risk (s)

Based on these two factors, the priority of each risk can be computed as follows:

p = r X s

where, p is the priority with which the risk must be handled, r is the probability of the risk becoming real, and s is the severity of damage caused due to the risk becoming real.

Risk Mitigation

After all the identified risks of a project have been assessed, plans are made to contain the most damaging and the most likely risks first. Different types of risks require different containment procedures. There are **three** main strategies for risk containment:

Avoid the risk: Risks can be avoided in several ways. Risks often arise due to project constraints and can be avoided by suitably modifying the constraints. The different categories of constraints that usually give rise to risks are:

- **Process-related risk**: These risks arise due to aggressive work schedule, budget, and resource utilization.
- **Product-related risks**: These risks arise due to commitment to challenging product features (e.g. response time of one second, etc.), quality, reliability etc.
- **Technology-related risks:** These risks arise due to commitment to use certain technology (e.g., satellite communication).

A **few examples** of risk avoidance can be the following: *Discussing with the customer to change the requirements to reduce the scope of the work, giving incentives to the developers to avoid the risk of manpower turnover, etc.*

Transfer the risk: This strategy involves getting the risky components developed by a third party, buying insurance cover, etc.

Risk reduction: This involves planning ways to contain the damage due to a risk. The most important risk reduction techniques for technical risks is to build a prototype that tries out the technology that you are trying to use.

The project manager must consider the cost of handling the risk and the corresponding reduction of risk. For this we may compute the **risk leverage** of the different risks. Risk leverage is the difference in risk exposure divided by the cost of reducing the risk.

$$\text{risk leverage} = \frac{\text{risk exposure before reduction} - \text{risk exposure after reduction}}{\text{cost of reduction}}$$

1.9. SOFTWARE CONFIGURATION MANAGEMENT

The configuration of the software is the state of all project deliverables at any point of time; and software configuration management deals with effectively tracking and controlling the configuration of a software during its life cycle.

There are several reasons for putting an object under configuration management. **If configuration management is not used**, every software developer has a personal copy of an object (e.g. source code). When a developer makes changes to his local copy, he is expected to intimate the changes that he has made to other developers, so that the necessary changes in interfaces could be uniformly carried out across all modules. However, not only would it eat up valuable time of the developers, but many times a developer might make changes to the interfaces in his local copies and forgets to intimate the teammates about the changes. This makes the different copies of the object inconsistent. Finally, when the different modules are integrated, it does not work. Therefore, when several team members work on developing an application, it is necessary for them to work on a single copy of the application, otherwise inconsistencies may arise

Configuration Management Activities

Configuration management is carried out through **two** principal activities:

Configuration identification: It involves deciding which parts of the system should be kept track of. Project managers normally classify the objects associated with a software development into **three** main categories—**controlled, precontrolled, and uncontrolled**. **Controlled objects** are those that are already under configuration control. The team members must follow some formal procedures to change them. **Precontrolled objects** are not yet under configuration control, but will eventually be under configuration control. **Uncontrolled** objects are not subject to configuration control. Controllable objects include both controlled and precontrolled objects. Typical controllable objects include:

- ✓ Requirement's specification document

- ✓ Design documents
- ✓ Tools used to build the system, such as compilers, linkers, lexical analysers, parsers,
- ✓ Source code for each module
- ✓ Test cases
- ✓ Problem reports

Configuration control: Configuration control allows only authorized changes to the controlled objects to occur and prevents unauthorized changes. The configuration control part of a configuration management system that most directly affects the day-to-day operations of developers. In order to change a controlled object such as a module, a developer can get a **private copy** of the module by a reserve operation. Configuration management tools allow only one person to reserve a module at any time. Once an object is reserved, it does not allow anyone else to reserve this module until the reserved module is restored. Thus, by preventing more than one developer to simultaneously reserve a module, the problems associated with concurrent access are solved.

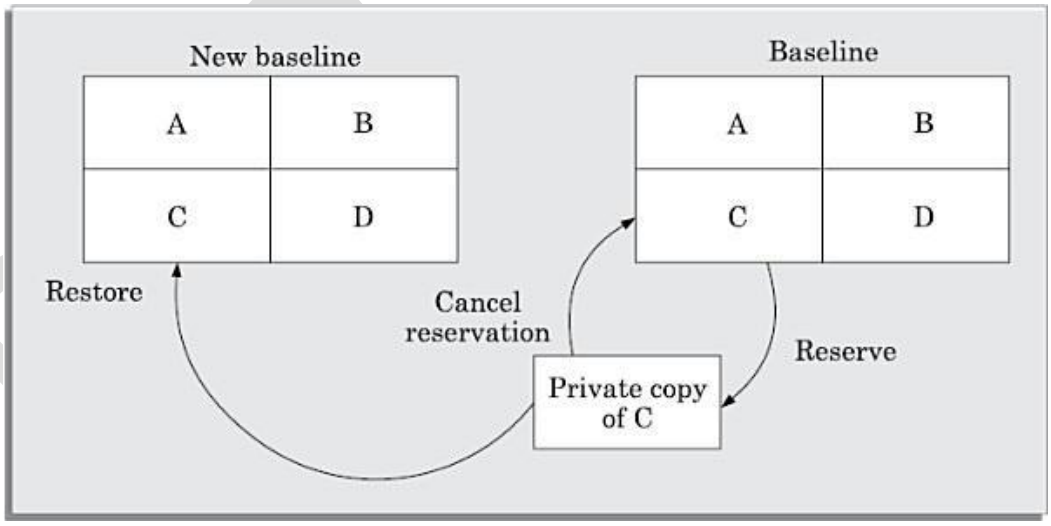


Figure: Reserve and restore operation in configuration control