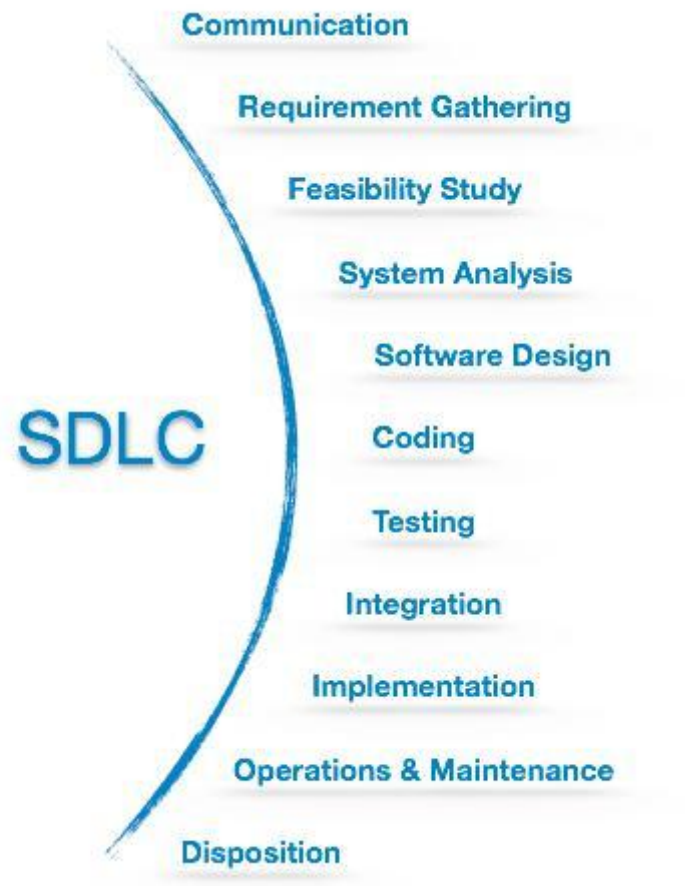
MODULE - I

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.

Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the intended software product.

SDLC Activities

SDLC provides a series of steps to be followed to design and develop a software product efficiently. SDLC framework includes the following steps:



Communication

This is the first step where the user initiates the request for a desired software product. He contacts the service provider and tries to negotiate the terms. He submits his request to the service providing organization in writing.

Requirement Gathering

This step onwards the software development team works to carry on the project. The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are contemplated and segregated into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given -

- studying the existing or obsolete system and software,
- conducting interviews of users and developers,
- referring to the database or
- collecting answers from the questionnaires.

Feasibility Study

After requirement gathering, the team comes up with a rough plan of software process. At this step the team analyzes if a software can be made to fulfill all requirements of the user and if there is any possibility of software being no more useful. It is found out, if the project is financially, practically and technologically feasible for the organization to take up. There are many algorithms available, which help the developers to conclude the feasibility of a software project.

System Analysis

At this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes Understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc. The project team analyzes the scope of the project and plans the schedule and resources accordingly.

Software Design

Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are the inputs of this step. The output of this step comes in the form of two designs; logical design and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams and in some cases pseudo codes.

Coding

This step is also known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.

Testing

An estimate says that 50% of whole software development process should be tested. Errors may ruin the software from critical level to its own removal. Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing and testing the product at user's end. Early discovery of errors and their remedy is the key to reliable software.

Integration

Software may need to be integrated with the libraries, databases and other program(s). This stage of SDLC is involved in the integration of software with outer world entities.

Implementation

This means installing the software on user machines. At times, software needs post-installation configurations at user end. Software is tested for portability and adaptability and integration related issues are solved during implementation.

Operation and Maintenance

This phase confirms the software operation in terms of more efficiency and less errors. If required, the users are trained on, or aided with the documentation on how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the changes taking place in user end environment or technology. This phase may face challenges from hidden bugs and real-world unidentified problems.

Disposition

As time elapses, the software may decline on the performance front. It may go completely obsolete or may need intense upgradation. Hence a pressing need to eliminate a major portion of the system arises. This phase includes archiving data and required software components, closing down the system, planning disposition activity and terminating system at appropriate end-of-system time.

CLASSICAL WATERFALL MODEL --

The classical waterfall model is intuitively the most obvious way to develop software. Though the classical waterfall model is elegant and intuitively obvious, it is not a practical model in the sense that it cannot be used in actual software development projects. Thus, this model can be considered to be a *theoretical way of developing software*. But all other life cycle models are essentially derived from the classical waterfall model. So, in order to be able to appreciate other life cycle models it is necessary to learn the classical waterfall model. Classical waterfall model divides the life cycle into the following phases as shown in fig.:

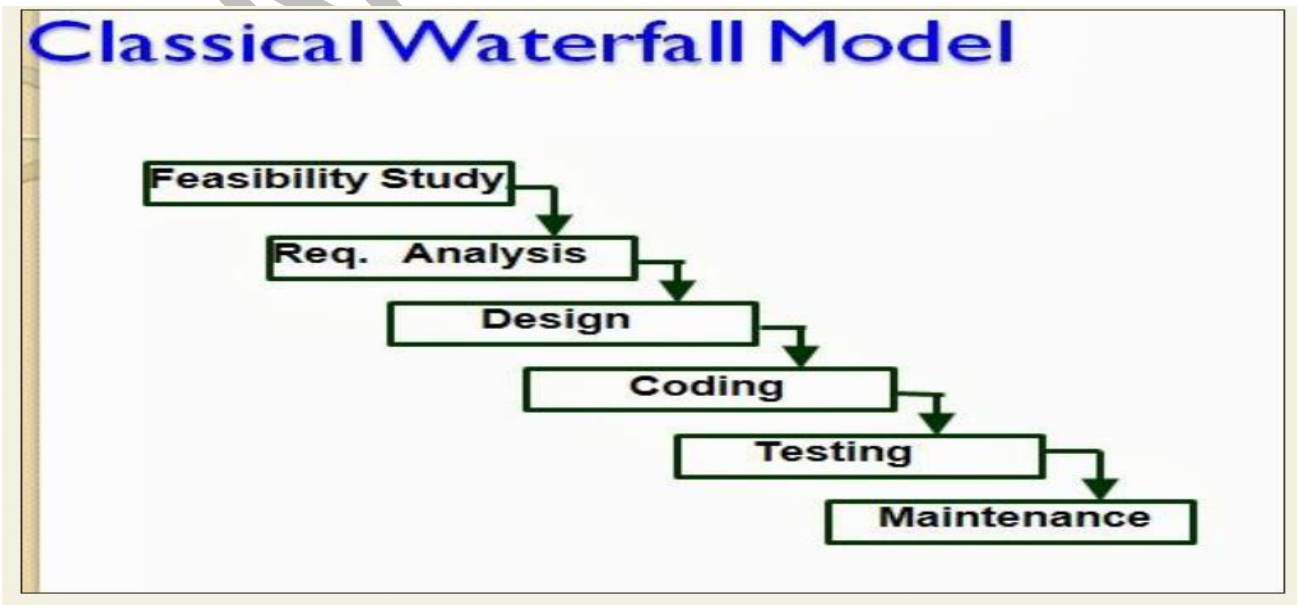


Fig : Classical Waterfall Model

Feasibility study - The main aim of feasibility study is to determine whether it would be financially and technically feasible to develop the product.

At first project managers or team leaders try to have a rough understanding of what is required to be done by visiting the client side. They study different input data to the system and output data to be produced by the system. They study what kind of processing is needed to be done on these data and they look at the various constraints on the behavior of the system.

- After they have an overall understanding of the problem they investigate the different solutions that are possible. Then they examine each of the solutions in terms of what kind of resources required, what would be the cost of development and what would be the development time for each solution.
- Based on this analysis they pick the best solution and determine whether the solution is feasible financially and technically. They check whether the customer budget would meet the cost of the product and whether they have sufficient technical expertise in the area of development.

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
- Requirements specification

The goal of the requirements gathering activity is to collect all relevant information from the customer regarding the product to be developed. This is done to clearly understand the customer requirements so that incompleteness and inconsistencies are removed.

The requirements analysis activity is begun by collecting all relevant data regarding the product to be developed from the users of the product and from the customer through interviews and discussions. For example, to perform the requirements analysis of a business accounting software required by an organization, the analyst might interview all the accountants of the organization to ascertain their requirements. The data collected from such a group of users usually contain several contradictions and ambiguities, since each user typically has only a partial and incomplete view of the system. Therefore it is necessary to identify all ambiguities and contradictions in the requirements and resolve them through further discussions with the customer. After all ambiguities, inconsistencies, and incompleteness have been resolved and all the requirements properly understood, the requirements specification activity can start. During this activity, the user requirements are systematically organized into a Software Requirements Specification (SRS) document. The customer requirements identified during the requirements gathering and analysis activity are organized into a SRS document. The important components of this document are functional requirements, the nonfunctional requirements, and the goals of implementation.

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 approaches are available: the traditional design approach and the object-oriented design approach.

 Traditional design approach -Traditional design consists of two different activities; first a 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 activity. During structured design, 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 phase (sometimes called the implementation phase) of software development is to translate the software design into source code. Each component of the design is implemented as a program module. The end-product of this phase is a set of program modules that have been individually tested. During this phase, each module is unit tested to determine the correct working of all the individual modules. It involves testing each module in isolation as this is the most efficient way to debug the errors identified at this stage.

Integration and system testing: -Integration of different modules is undertaken once they have been coded and unit tested. During the integration and system testing phase, the modules are integrated in a planned manner. The different modules making up a software product are almost never integrated in one shot. Integration is normally carried out incrementally over a number of steps. During each integration step, the partially integrated system is tested and a set of previously planned modules are added to it. Finally, when all the modules have been successfully integrated and tested, system testing is carried out. The goal of system testing is to ensure that the developed system conforms to its requirements laid out in the SRS document. System testing usually consists of three different kinds of testing activities:

 α– testing: It is the system testing performed by the development team.

 β–testing: It is the system testing performed by a friendly set of customers.

 Acceptance testing: It is the system testing performed by the customer himself after the product delivery to determine whether to accept or reject the delivered product.

System testing is normally carried out in a planned manner according to the system test plan document. The system test plan identifies all testing-related activities that must be performed,

specifies the schedule of testing, and allocates resources. It also lists all the test cases and the expected outputs for each test case.

Maintenance: -Maintenance of a typical software product requires much more than the effort necessary to develop the product itself. Many studies carried out in the past confirm this and indicate that the relative effort of development of a typical software product to its maintenance effort is roughly in the 40:60 ratios. Maintenance involves performing any one or more of the following three kinds of activities:

 Correcting errors that were not discovered during the product development phase. This is called corrective maintenance.

 Improving the implementation of the system, and enhancing the functionalities of the system according to the customer's requirements. This is called perfective maintenance.

 Porting the software to work in a new environment. For example, porting may be required to get the software to work on a new computer platform or with a new operating system. This is called adaptive maintenance.

Shortcomings of the classical waterfall model

The classical waterfall model is an idealistic one since it assumes that no development error is ever committed by the engineers during any of the life cycle phases. However, in practical development environments, the engineers do commit a large number of errors in almost every phase of the life cycle. The source of the defects can be many: oversight, wrong assumptions, use of inappropriate technology, communication gap among the project engineers, etc. These defects usually get detected much later in the life cycle. For example, a design defect might go unnoticed till we reach the coding or testing phase. Once a defect is detected, the engineers need to go back to the phase where the defect had occurred and redo some of the work done during that phase and the subsequent phases to correct the defect and its effect on the later phases. Therefore, in any practical software development work, it is not possible to strictly follow the classical waterfall model.

ITERATIVE WATERFALL MODEL--

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.

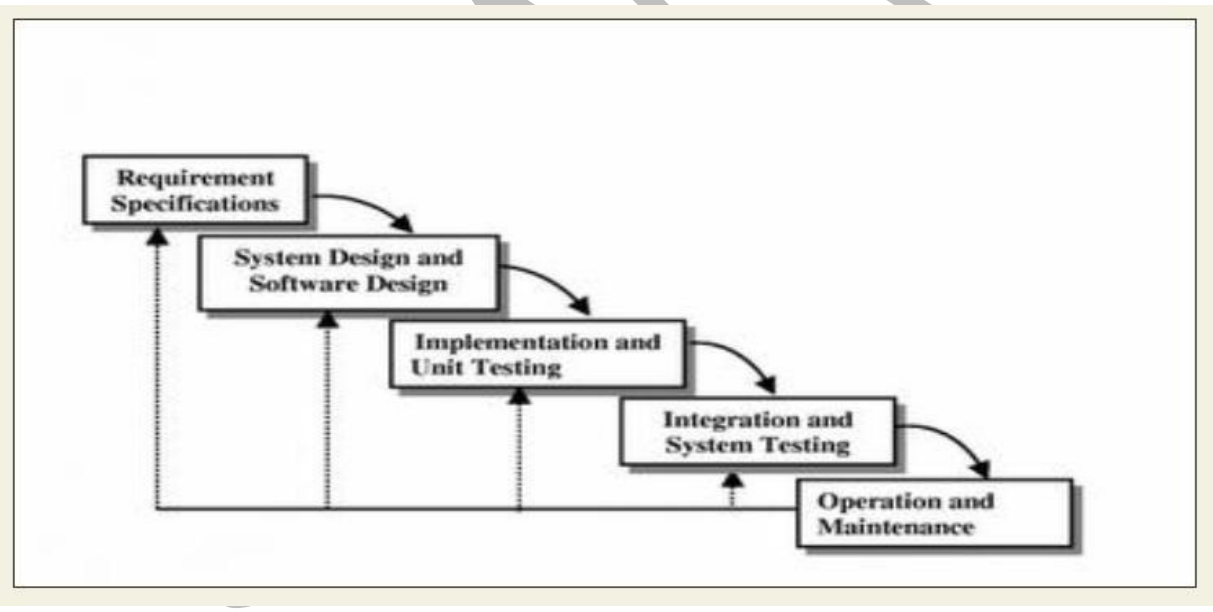


Fig : Iterative Waterfall Model

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget.

The disadvantage with this SDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

PRTOTYPING MODEL--

Prototype

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

Need for a prototype in software development

There are several uses of a prototype. An important purpose is to illustrate the input data formats, messages, reports, and the interactive dialogues to the customer. This is a valuable mechanism for gaining better understanding of the customer's needs:

- how the screens might look like
- how the user interface would behave
- how the system would produce outputs

Another reason for developing a prototype is that it is impossible to get the perfect product in the first attempt. Many researchers and engineers advocate that if you want to develop a good product you must plan to throw away the first version. The experience gained in developing the prototype can be used to develop the final product.

A prototyping model can be used when technical solutions are unclear to the development team. A developed prototype can help engineers to critically examine the technical issues associated with the product development. Often, major design decisions depend on issues like the response time of a hardware controller, or the efficiency of a sorting algorithm, etc. In such circumstances, a prototype may be the best or the only way to resolve the technical issues.

A prototype of the actual product is preferred in situations such as:

• User requirements are not complete

• Technical issues are not clear

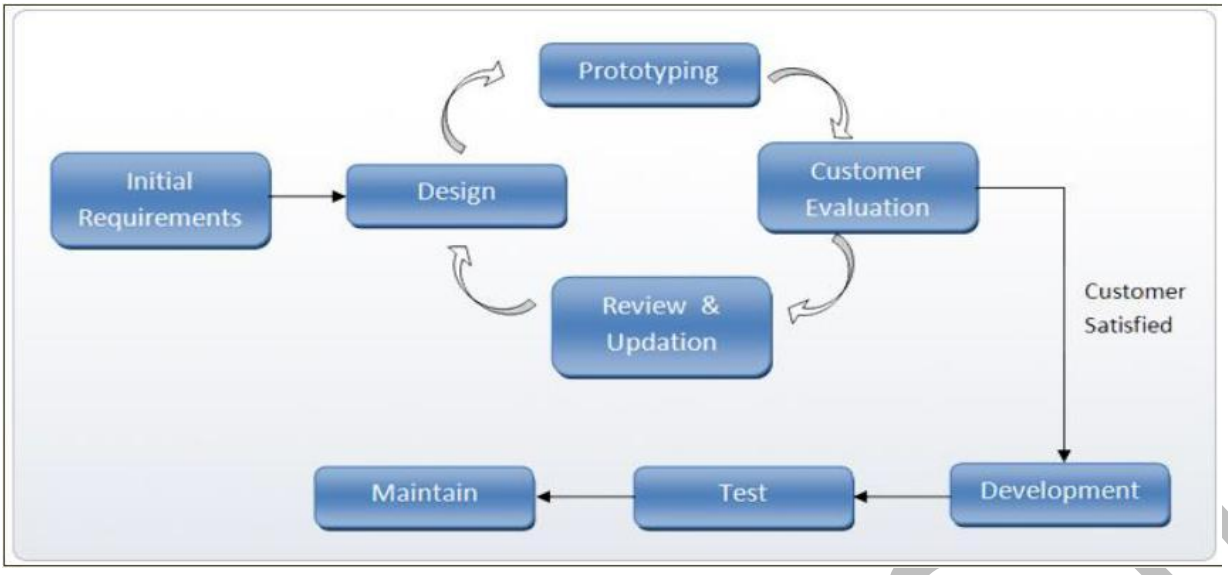


Fig : Prototype Model

4. EVOLUTIONARY MODEL--

It is also called *successive versions model* or *incremental model*. At first, a simple working model is built. Subsequently it undergoes functional improvements & we keep on adding new functions till the desired system is built.

Applications:

- Large projects where you can easily find modules for incremental implementation. Often used when the customer wants to start using the core features rather than waiting for the full software.
- Also used in object oriented software development because the system can be easily portioned into units in terms of objects.

Advantages:

- User gets a chance to experiment partially developed system
- Reduce the error because the core modules get tested thoroughly.

Disadvantages:

- It is difficult to divide the problem into several versions that would be acceptable to the customer which can be incrementally implemented & delivered.

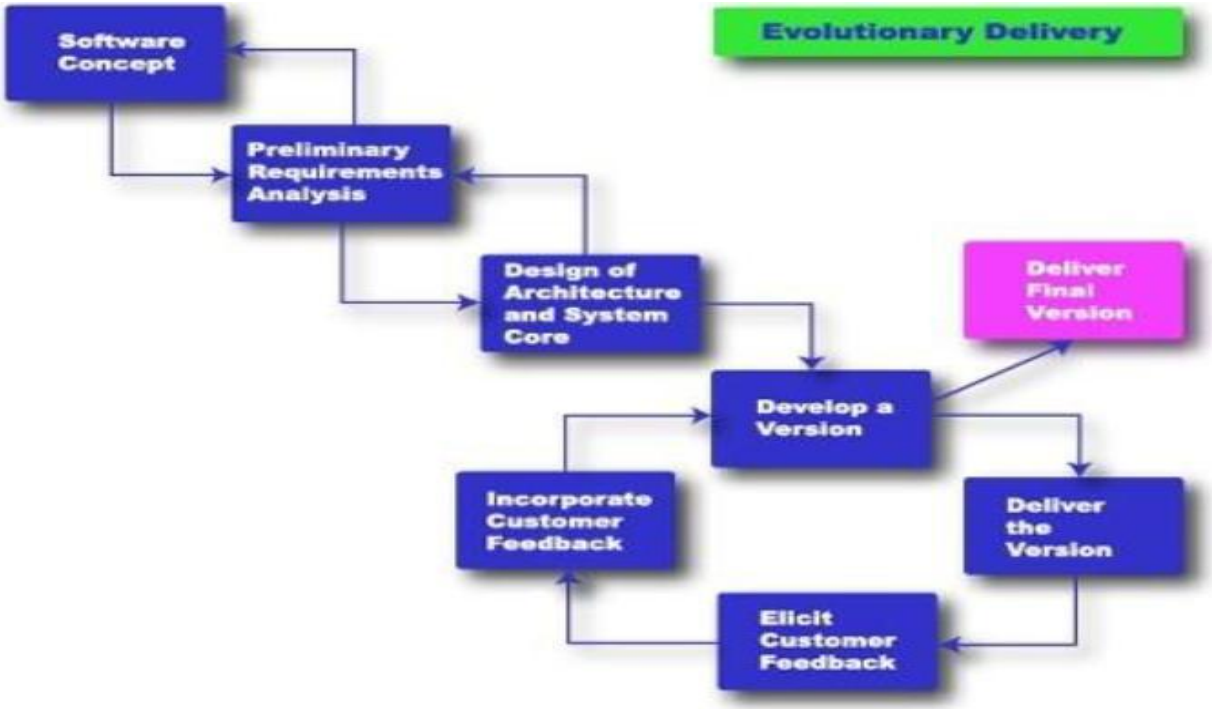


Fig : Evolutionary Model

SPIRAL MODEL--

The Spiral model of software development is shown in figure.The diagrammatic representation of this model appears like a spiral with many loops. The exact number of loops in the spiral is not fixed. Each loop of the spiral represents a phase of the software process. For example, the innermost loop might be concerned with feasibility study, the next loop with requirements specification, the next one with design, and so on. Each phase in this model is split into four sectors (or quadrants) as shown in figure. The following activities are carried out during each phase of a spiral model.

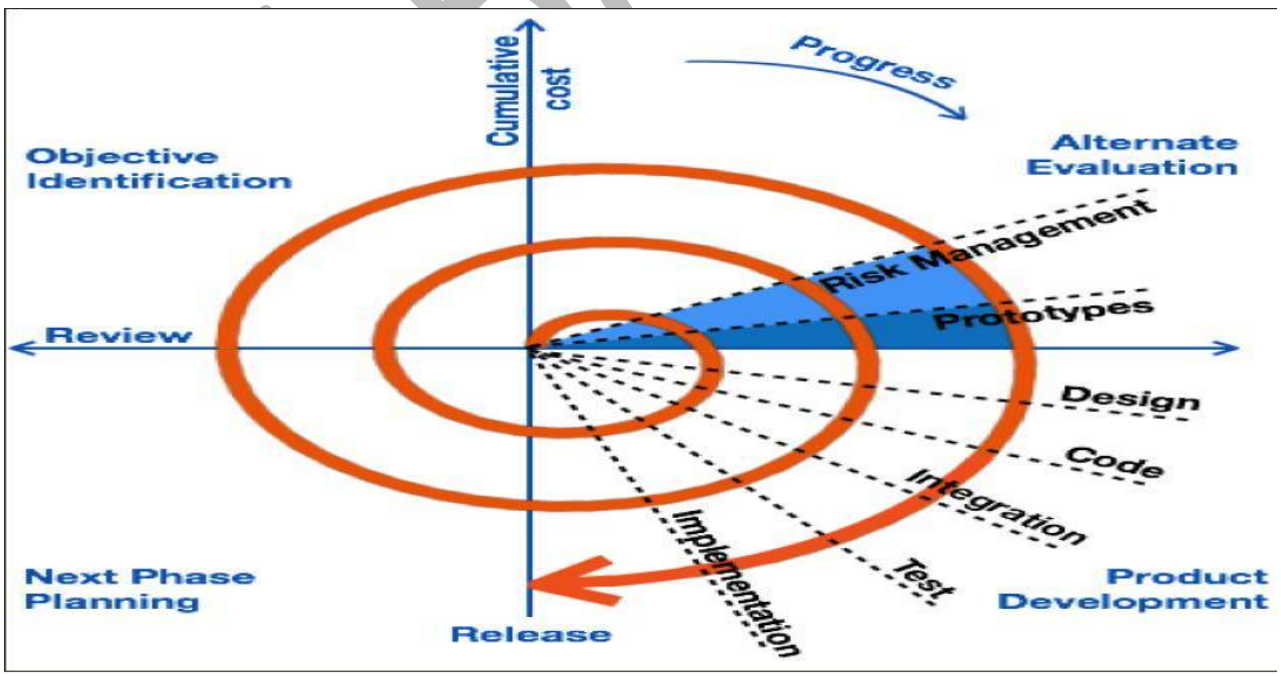


Fig : Spiral Model

First quadrant (Objective Setting)

• During the first quadrant, it is needed to identify the objectives of the phase.
• Examine the risks associated with these objectives.

Second Quadrant (Risk Assessment and Reduction)
• A detailed analysis is carried out for each identified project risk.
• Steps are taken to reduce the risks. For example, if there is a risk that the requirements are inappropriate, a prototype system may be developed.

Third Quadrant (Development and Validation)
• Develop and validate the next level of the product after resolving the identified risks.

Fourth Quadrant (Review and Planning)
• Review the results achieved so far with the customer and plan the next iteration around the spiral.
• Progressively more complete version of the software gets built with each iteration around the spiral.

Circumstances to use spiral model

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

Comparison of different life-cycle models

The classical waterfall model can be considered as the basic model and all other life cycle models as embellishments of this model. However, the classical waterfall model cannot be used in practical development projects, since this model supports no mechanism to handle the errors committed during any of the phases.

This problem is overcome in the iterative waterfall model. The iterative waterfall model is probably the most widely used software development model evolved so far. This model is simple to understand and use. However this model is suitable only for well-understood problems; it is not suitable for very large projects and for projects that are subject to many risks.

The prototyping model is suitable for projects for which either the user requirements or the underlying technical aspects are not well understood. This model is especially popular for development of the user-interface part of the projects.

The evolutionary approach is suitable for large problems which can be decomposed into a set of modules for incremental development and delivery. This model is also widely used for object-oriented development projects. Of course, this model can only be used if the incremental delivery of the system is acceptable to the customer.

The spiral model is called a meta model since it encompasses all other life cycle models. Risk handling is inherently built into this model. The spiral model is suitable for development of technically challenging software products that are prone to several kinds of risks. However, this model is much more complex than the other models – this is probably a factor deterring its use in ordinary projects.

The different software life cycle models can be compared from the viewpoint of the customer. Initially, customer confidence in the development team is usually high irrespective of the development model followed. During the lengthy development process, customer

confidence normally drops off, as no working product is immediately visible. Developers answer customer queries using technical slang, and delays are announced. This gives rise to customer resentment. On the other hand, an evolutionary approach lets the customer experiment with a working product much earlier than the monolithic approaches. Another important advantage of the incremental model is that it reduces the customer's trauma of getting used to an entirely new system. The gradual introduction of the product via incremental phases provides time to the customer to adjust to the new product. Also, from the customer's financial viewpoint, incremental development does not require a large upfront capital outlay. The customer can order the incremental versions as and when he can afford them.

The RAD Model--

Rapid Application Development (RAD) is an incremental software process model that emphasizes a short development cycle. The RAD model is a "high-speed" adaptation of the waterfall model, in which rapid development is achieved by using a component-based construction approach. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a "fully functional system" within a very short time period (e.g., 60 to 90 days) .

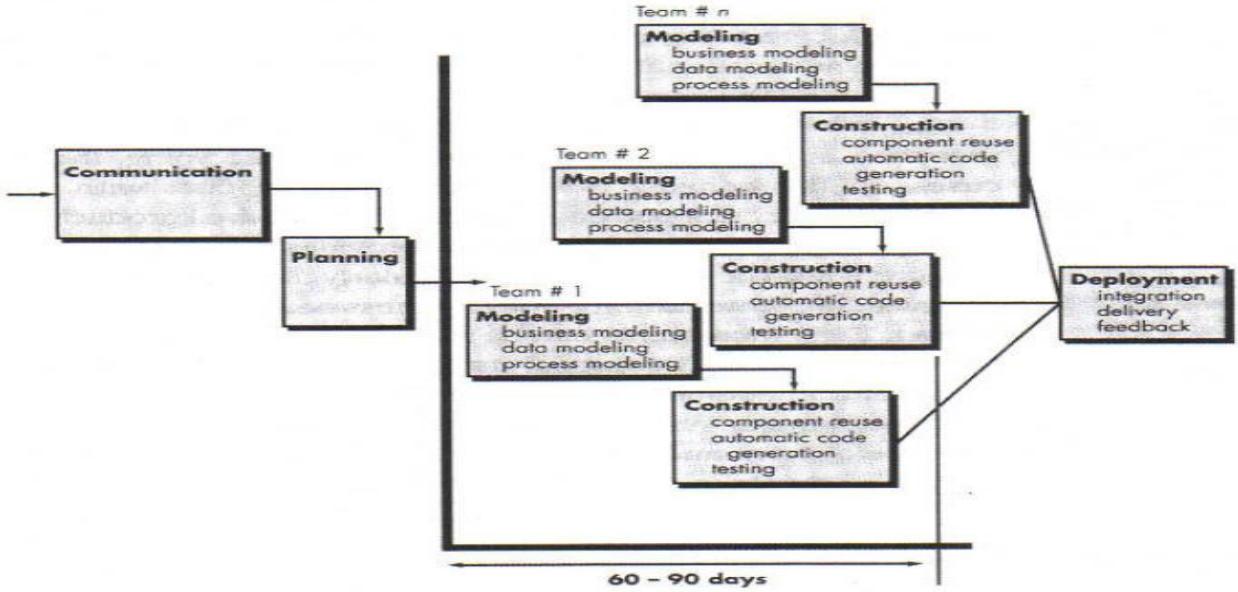
Like other process models, the RAD approach maps into the generic framework activities. Communication works to understand the business problem and the information characteristics that the software must accommodate. Planning is essential because multiple software teams work in parallel on different system functions. Modeling encompasses three major phases—business modeling, data modeling and process modeling—and establishes design representations that serve as the basis for RAD's construction activity. Construction emphasizes the use of preexisting software components and the application of automatic code generation. Finally, deployment establishes a basis for subsequent iterations, if required

The RAD process model is illustrated in Figure.

Obviously, the time constraints imposed on a RAD project demand "scalable scope". If a business application can be modularized in a way that enables each major function to be completed in less than three months (using the approach described above), it is a candidate for RAD. Each major function can be addressed by a separate RAD team and then integrated to form a whole.

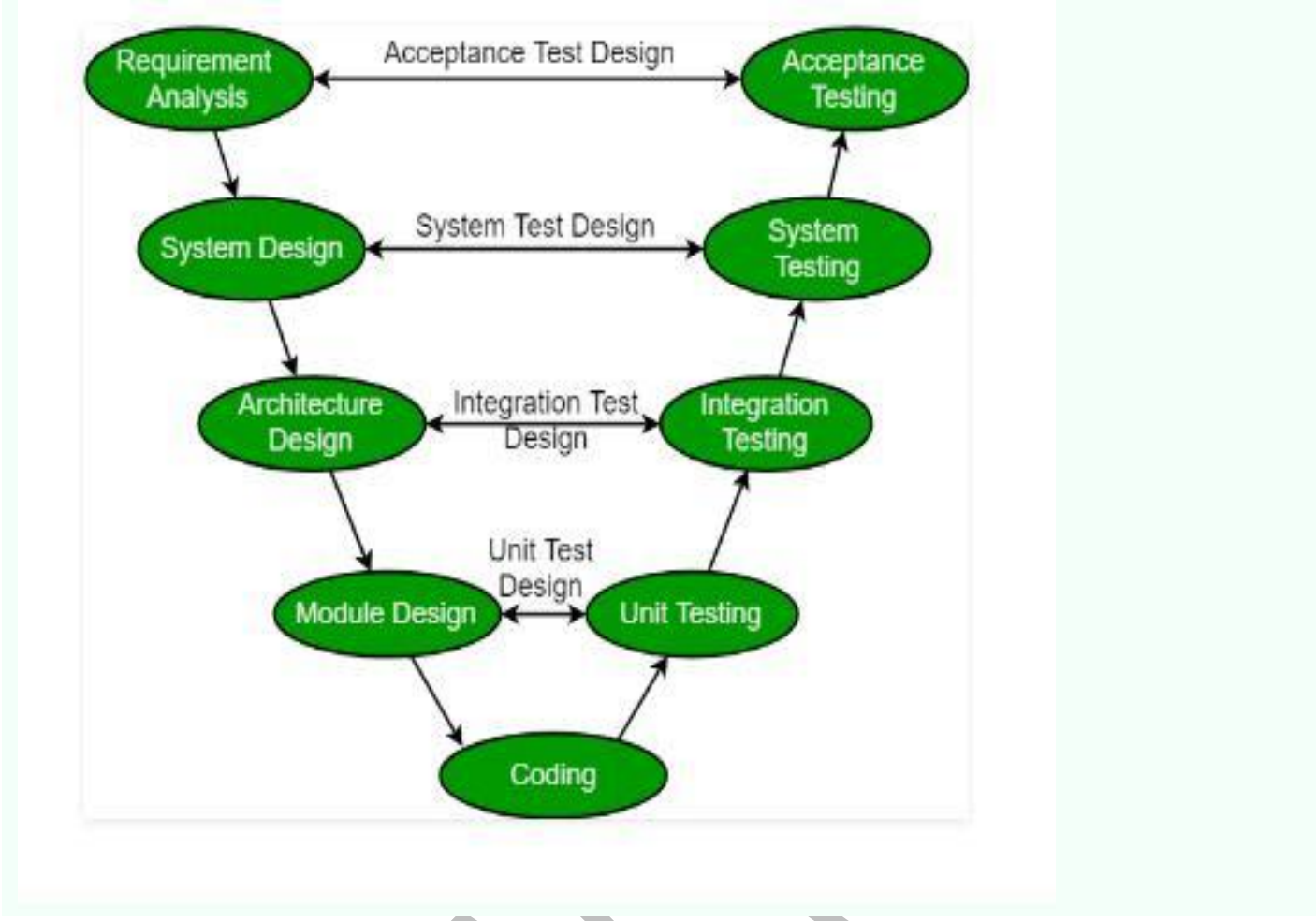
Like all process models, the RAD approach has drawbacks : (1) for large, but scalable projects, RAD requires sufficient human resources to create the right number of RAD teams; (2) if developers and customers are not committed to the rapid-fire activities necessary to complete the system in a much abbreviated time frame, RAD projects will fail; (3) if a system cannot be properly modularized, building the components necessary for RAD will be problematic; (4) if high performance is an issue, and performance is to be

achieved through tuning the interfaces to system components, the RAD approach may not work; and (5) RAD may not be appropriate when technical risks are high (e.g., when a new application makes heavy use of new technology).



V-Model

The V-model is a type of SDLC model where process executes in a sequential manner in V-shape. It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.



Verification: It involves static analysis technique (review) done without executing code. It is the process of evaluation of the product development phase to find whether specified requirements meet.
Validation: It involves dynamic analysis technique (functional, non-functional), testing done by executing code. Validation is the process to evaluate the software after the completion of the development phase to determine whether software meets the customer expectations and requirements.
So V-Model contains Verification phases on one side of the Validation phases on the other side. Verification and Validation phases are joined by coding phase in V-shape. Thus it is called V-Model.

Design Phase:

- **Requirement Analysis:** This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
- **System Design:** This phase contains the system design and the complete hardware and communication setup for developing product.
- **Architectural Design:** System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
- **Module Design:** In this phase the system breaks dowm into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

Testing Phases:

- **Unit Testing:** Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
- **Integration testing:** After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.
- **System Testing:** System testing test the complete application with its functionality, inter dependency, and communication.It tests the functional and non-functional requirements of the developed application.
- **User Acceptance Testing (UAT):** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user's requirement and system is ready for use in real world.

Principles of V-Model:

- **Large to Small:** In V-Model, testing is done in a hierarchical perspective, For example, requirements identified by the project team, create High-Level Design, and Detailed Design phases of the project. As each of these phases is completed the requirements, they are defining become more and more refined and detailed.
- **Data/Process Integrity:** This principle states that the successful design of any project requires the incorporation and cohesion of both data and processes. Process elements must be identified at each and every requirements.
- **Scalability:** This principle states that the V-Model concept has the flexibility to accommodate any IT project irrespective of its size, complexity or duration.
- **Cross Referencing:** Direct correlation between requirements and corresponding testing activity is known as cross-referencing.
- **Tangible Documentation:** This principle states that every project needs to create a document. This documentation is required and applied by both the project development team and the support team. Documentation is used to maintaining the application once it is available in a production environment.

Advantages:

- This is a highly disciplined model and Phases are completed one at a time.
- V-Model is used for small projects where project requirements are clear.
- Simple and easy to understand and use.
- This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
- It enables project management to track progress accurately.

Disadvantages:

- High risk and uncertainty.
- It is not a good for complex and object-oriented projects.
- It is not suitable for projects where requirements are not clear and contains high risk of changing.
- This model does not support iteration of phases.
- It does not easily handle concurrent events.

Agile Models—

• Extreme Programming (XP)

• Adaptive Software Development (ASD)

Extreme Programming

• It is most widely used agile process model.

• XP uses an object-oriented approach as its preferred development paradigm.

• It defines four (4) framework activities Planning, Design, Coding, and Testing.

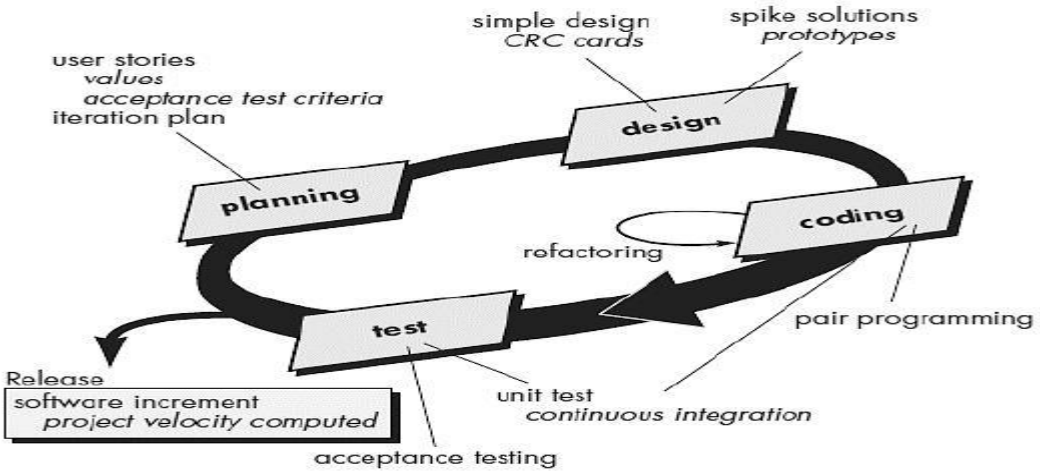


Figure: Extreme Programming Process

Figure: Extreme Programming Process

Planning:

• Begins with the creation of a set of stories (also called user stories)

• Each story is written by the customer and is placed on an index card

• The customer assigns a value (i.e. a priority) to the story

• Agile team assesses each story and assigns a cost

• Stories are grouped to for a deliverable increment

• A commitment is made on delivery date

• After the first increment "project velocity" is used to help define subsequent delivery dates for other increments

Design:

• Follows the keep it simple principle

• Encourage the use of CRC (class-responsibility-collaborator) cards

• For difficult design problems, suggests the creation of "spike solutions"—a design prototype

• Encourages "refactoring"—an iterative refinement of the internal program design

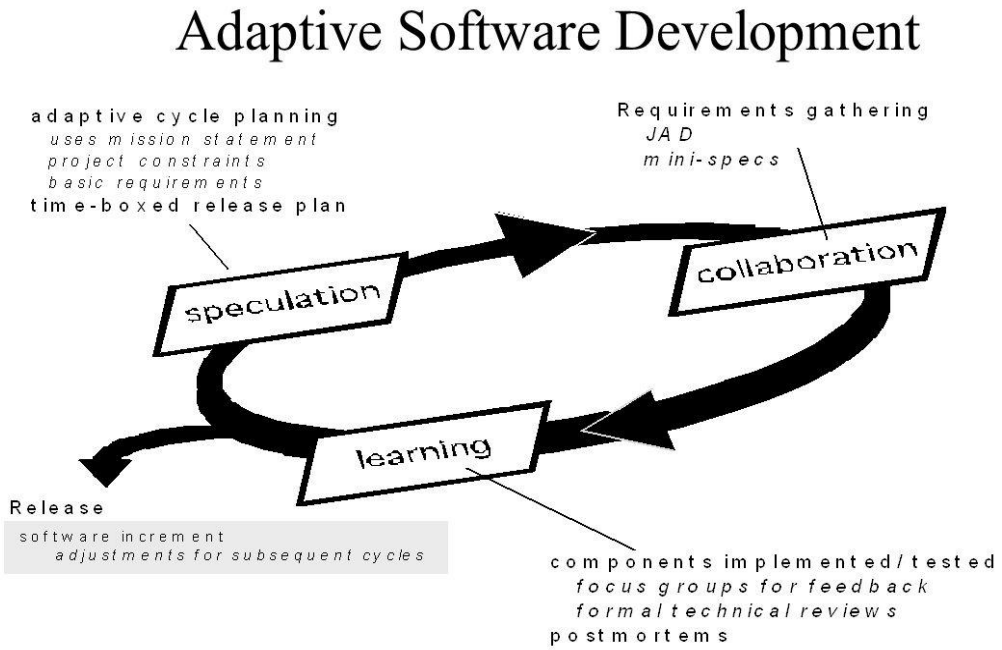
• Design occurs both before and after coding commences

Coding:

• Recommends the construction of a series of unit tests for each of the stories before coding commences

• Encourages "pair programming"
— Developers work in pairs, checking each other's work and providing the support to always do a good job.
— Mechanism for real-time problem solving and real-time quality assurance
— Keeps the developers focused on the problem at hand
• Needs continuous integration with other portions (stories) of the s/w, which provides a "smoke testing" environment
Testing:
• Unit tests should be implemented using a framework to make testing automated. This encourages a regression testing strategy.
• Integration and validation testing can occur on a daily basis
• Acceptance tests, also called customer tests, are specified by the customer and executed to assess customer visible functionality
• Acceptance tests are derived from user stories

Adaptive Software Development--



Adaptive software development (ASD) phases,

In ASD, projects happen in an iterative cycle composed of three overlapping phases: speculation, collaboration, and learning. The "speculation" phase is what is usually called "planning" in other methodologies, agile or not. It explicitly acknowledges the paradox of making plans in a rapid-evolving, complex, scenario.
The collaboration phase shows the importance of closer and constant collaboration, not only within the development team, but also between developers and the end-users of the software.
Finally, the learning phase indicates everyone involved in the software development process will be continuously learning throughout the project.

ASD, as an approach, is oriented toward high-speed: it uses techniques such as time-boxed iterative cycles, risk-driven planning and concurrency to achieve quick delivery of value to the customer, while also embracing the uncertainty and near-chaos that are intrinsic to complex, high-risk projects.

How does it work?

In ASD, as in most agile methodologies, work occurs in cycles or iterations. During such cycles, developers not only build new components but also make necessary modifications to pre-existing ones.

A key distinction between adaptive software development and other methodologies is that, in ASD, cycles are component-based, rather than task-based. It is common, for instance, for teams using other methodologies to break down user stories into more granular tasks, which are then assigned to developers to implement. In ASD, the focus is always the desired result. It defines components as a group of features, planned, implemented and delivered together.

Components, in the ASD sense, don't refer only to visual components—such as buttons, form fields, or other GUI elements. It also includes all related "things" that are implemented together.

For instance, an online payment service could have an "invoice management" component, which would include not only the graphical elements necessary to carry out those tasks, but also the underlying APIs and storage mechanisms.

More specifically, there are three types of components in ASD: primary components, technology components, and support components. Primary components refer to the business functionalities themselves. Technology components, on the other hand, consist of pieces of technology or infrastructure that need to be in place for the primary components to be implemented. Technology components include not only hardware and network infrastructure, but also software such as operating systems, databases, frameworks, and more. What will often happen is that such components will already be ready to use. If they're not, they need to be assigned to cycles for installation.

Support components include everything else not contemplated by the other two types. For instance, external-facing documentation, training materials, and more.

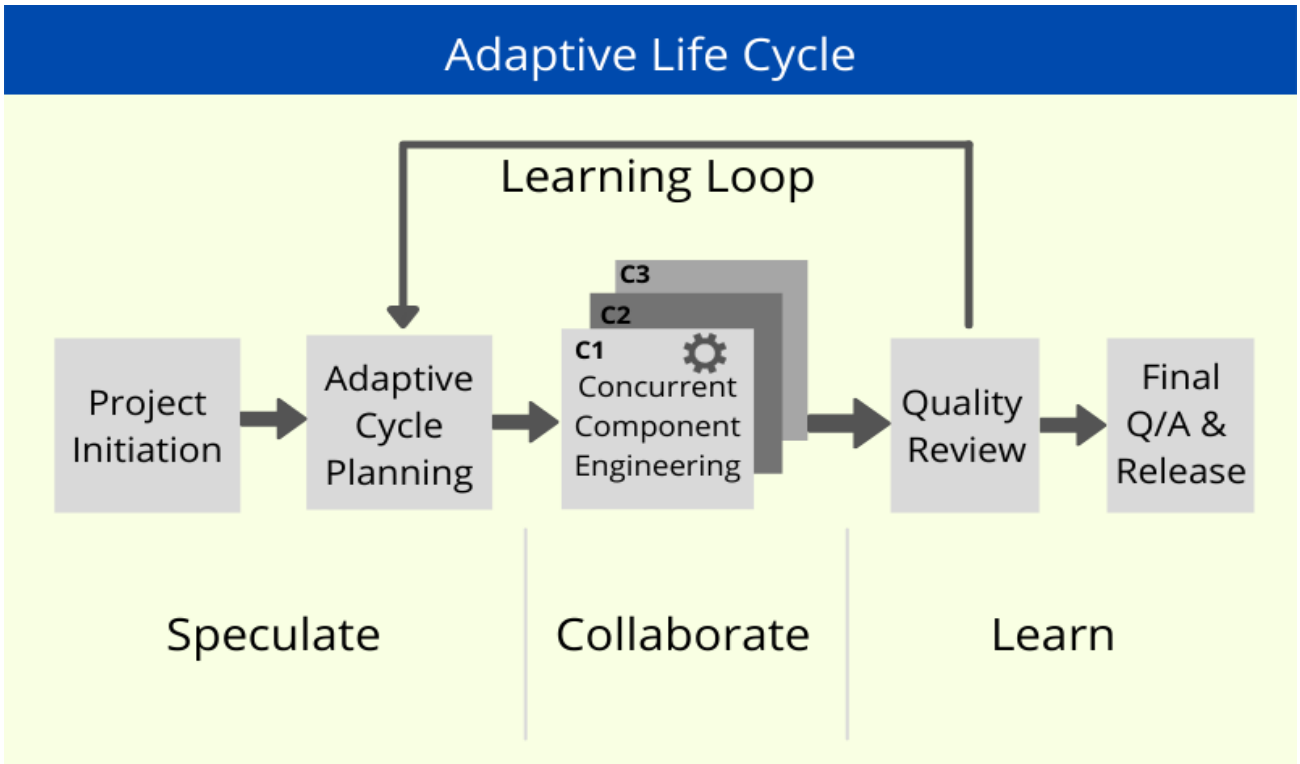
As in other methodologies, cycles in ASD are time-boxed, not only at the level of each individual development cycle, but also at the project level. Finally, ASD cycles are risk-driven and change-tolerant.

Being risk-driven can be understood as a willingness to fail fast., by placing higher-risk items in the earlier cycles. If you identify a risk in your project, you should try to reduce the likelihood of the failure. However, if reducing the probability isn't possible, ASD says you should accelerate the happening of the failure. The rationale is simple: if a given decision is doomed to fail, it's better and cheaper to find out earlier rather than later.

Being change-tolerant means that, instead of assuming most activities will happen as planned, you start with the premise that everything will change, and often. In ASD, teams create change-tolerant environments so change can be better managed. One of the ways in which this is accomplished is through the adoption of shorter time-frames. Also, ASD promotes a culture in which developers are constantly asking themselves questions about change, and also watching and evaluating what competitor services and products are doing.

Instead of trying to avoid change, ASD embraces that for the benefit of the client.

The Adaptive Life Cycle--



The adaptive life cycle can be seen as an evolution of the evolutionary life cycle.

Speculate

The "speculate" phase in ASD addresses the inherent paradox which is planning in complex scenarios. Since you can't predict your outcomes, you're way more likely to be wrong than right during the planning phases.

ASD acknowledges that by getting rid of the word "planning"—which is a loaded term, carrying a lot of baggage—and replaces it with "speculating." When speculating, teams do the following:

- they define their mission statement to the best of their ability;
- then, they express a general idea of their end goals, which happens through the creation and sharing of mission artifacts;
- Finally, they adopt mechanisms to help them adapt and change as they learn throughout the lifetime of the project.

The mission artifacts in ASD are mainly three:

- The project vision charter. A simple visual summary for the project, which includes business outcomes and the main market differentiators.

- The project data sheet. A short document summarizing essential information for the project the team and customers can use as a point of focus.

- And the product specification outline. A high-level inventory of the product specifications, whose main goal is to define the expected components—group of features—for the product.

In practice, the speculation phase is divided into two steps: the project initiation step and the adaptive planning step.

The project initiation step is the earliest phase in which the organization develops project management information, a mission statement for the project and initial requirements.

The adaptive planning step is the phase in which software components are assigned for being built or assembled. During this phase, the team executes the following steps:

- They decide the time-box for the project
- Then, they decide the necessary number of development cycles, and how much time each cycle will take (4 to 8 eights for projects time-boxed to less than 10 months, and 6 to 10 weeks for projects longer than 9 months.)
- They specify a theme and objective for the cycle
- They assign components for each cycle
- Optionally, a task list is created for the project

The last step is optional because, ideally, adaptive management should forgo the need to control and micromanage, preferring to define the "what" and not care about the specifics of the "how". But since a transition to an adaptive style might be hard and even scary, the creation of a project task list might be used to smooth such a transition.

Collaborate

The collaborate phase is the phase in which developers actually perform the development work. This phase is divided into cycles, and each one of them delivers value to the customer.

What High-Performance Teams Look Like

A crucial and often overlooked characteristic of teams that perform well in complex projects is the high technical proficiency of each member. It doesn't matter if teams collaborate well and have great communication if the technical skills each participant brings to the team are not up to the challenge.

Additionally, teams should be small, with fewer than 10 members. That's vital because great collaboration requires intense interactions that get harder as teams get larger.

ASD advocates that a team should have all of the skills necessary to perform the work. This is common in other agile methodologies as well—e.g. the "whole team" concept from extreme programming. However, ASD stresses that the necessary blend of skills must include not only the technical chops but also interpersonal, and business skills as well.

The Core Values of Teams In ASD

In adaptive software development, teams operate according to the 4 core values: mutual trust, mutual respect, mutual participation, and mutual commitment.

Mutual Trust

Mutual trust must be achieved in the team through honesty—saying the truth; safety—the creation of a comfortable, welcoming environment for the expression of ideas; and dependability—knowing you can rely on your teammates.

Civility is a goal in great teams, but not necessarily harmony. Conflict is inevitable when there's diversity, and when handled correctly it boosts creativity.

Mutual Respect

In teams using adaptive software development, everyone must be respected and valued for the contributions they bring to the project, regardless of the areas in which they specialize.

Harmful behaviors and habits that are common in technical scenarios, must be abandoned if you are to achieve mutual respect. One of such behaviors is immediately dismissing someone as incompetent due to an opinion of theirs you happen to disagree with, and then refusing to accept their contributions from that point onwards.

Mutual Participation

Open, collaborative teams work best when more people participate. The value of mutual participation acknowledges that people have different levels of experience and knowledge, and that they have expertise in different areas.

The value doesn't advocate for equal participation, but for mutual participation. It accepts that each person on the team might contribute and express themselves according to their preferences and skills, but all participation is valued the same.

Bear in mind that the values build on each other. For instance, it's impossible to achieve a high degree of mutual participation if the team doesn't already have mutual trust in place. Trust is needed for team members to feel safe and comfortable expressing their ideas, without fear of judgment or ridicule.

Mutual Commitment

Members of the team are equally committed to the project's goal and they're all responsible for ensuring the project's success.

Even though successful software projects will have one or even more leaders, that doesn't mean the other team members are relieved from having responsibility or commitment to the end result. Quite the contrary: in adaptive software development, everyone owns the result, and abdicating your individual responsibility can and often does lead to disaster.

Learn

The learn stage in adaptive software development is essentially what other methodologies would call a review or retrospective. During this phase, the product developed is analyzed from a technical and also customer perspective. Additionally, the performance of the team is also reviewed, in search of opportunities for improvement.

SOFTWARE PROJECT MANAGEMENT

Responsibilities of a software project manager--

Software project managers take the overall responsibility of steering a project to success. It is very difficult to objectively describe the job responsibilities of a project manager. The job

responsibility of a project manager ranges from invisible activities like building up team morale to highly visible customer presentations. Most managers take responsibility for project proposal writing, project cost estimation, scheduling, project staffing, software process tailoring, project monitoring and control, software configuration management, risk management, interfacing with clients, managerial report writing and presentations, etc. These activities are certainly numerous, varied and difficult to enumerate, but these activities can be broadly classified into project planning, and project monitoring and control activities. The project planning activity is undertaken before the development starts to plan the activities to be undertaken during development. The project monitoring and control activities are undertaken once the development activities start with the aim of ensuring that the development proceeds as per plan and changing the plan whenever required to cope up with the situation.

Skills necessary for software project management

A theoretical knowledge of different project management techniques is certainly necessary to become a successful project manager. However, effective software project management frequently calls for good qualitative judgment and decision taking capabilities. In addition to having a good grasp of the latest software project management techniques such as cost estimation, risk management, configuration management, project managers need good communication skills and the ability get work done. However, some skills such as tracking and controlling the progress of the project, customer interaction, managerial presentations, and team building are largely acquired through experience. None the less, the importance of sound knowledge of the prevalent project management techniques cannot be overemphasized.

Project Planning

Once a project is found to be feasible, software project managers undertake project planning. Project planning is undertaken and completed even before any development activity starts. Project planning consists of the following essential activities:

 Project size: What will be problem complexity in terms of the effort and time required to develop the product?
 Cost: How much is it going to cost to develop the project?
 Duration: How long is it going to take to complete development?
 Effort: How much effort would be required?

The effectiveness of the subsequent planning activities is based on the accuracy of these estimations.

 Scheduling manpower and other resources.
 Staff organization and staffing plans.
 Risk identification, analysis, and abatement planning
 Miscellaneous plans such as quality assurance plan, configuration management plan, etc.

Precedence ordering among project planning activities

Different project related estimates done by a project manager have already been discussed. Figure shows the order in which important project planning activities may be undertaken. From figure it can be easily observed that size estimation is the first activity. It is also the most fundamental parameter based on which all other planning activities are carried out. Other estimations such as estimation of effort, cost, resource, and project duration are also very important components of project planning.

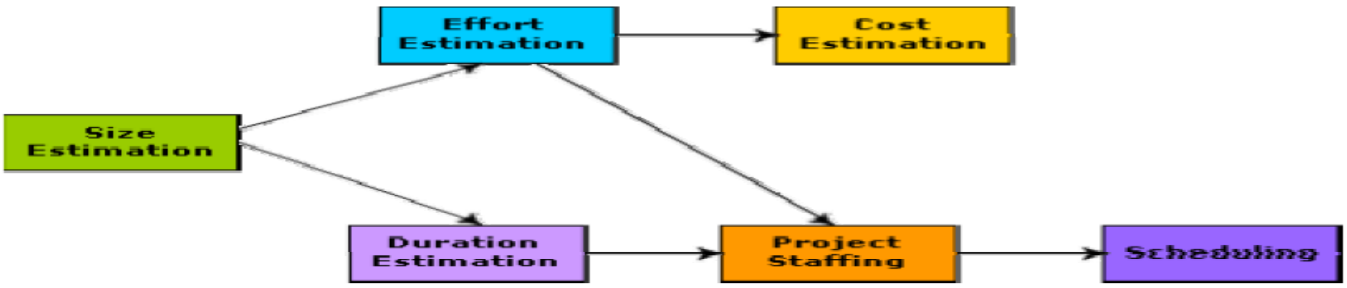


Fig. Precedence ordering among planning activities

Sliding Window Planning

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. However, project planning is a very challenging activity. Especially for large projects, it is very much difficult to make accurate plans. A part of this difficulty is due to the fact that the proper parameters, scope of the project, project staff, etc. may change during the span of the project. In order to overcome this problem, sometimes project managers undertake project planning in stages. Planning a project over a number of stages protects managers from making big commitments too early. This technique of staggered planning is known as Sliding Window Planning. In the sliding window technique, starting with an initial plan, the project is planned more accurately in successive development stages. At the start of a project, project managers have incomplete knowledge about the details of the project. Their information base gradually improves as the project progresses through different phases. After the completion of every phase, the project managers can plan each subsequent phase more accurately and with increasing levels of confidence.

Software Project Management Plan (SPMP)

Once project planning is complete, project managers document their plans in a Software Project Management Plan (SPMP) document. The SPMP document should discuss a list of different items that have been discussed below. 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**
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

METRICS FOR SOFTWARE PROJECT SIZE ESTIMATION

The project size is a measure of the problem complexity in terms of the effort and time required to develop the product. Currently two metrics are popularly being used widely to estimate size: lines of code (LOC) and function point (FP). The usage of each of these metrics in project size estimation has its own advantages and disadvantages.

Lines of Code (LOC)

LOC is the simplest among all metrics available to estimate project size. This metric is very popular because it is the simplest to use. Using this metric, the project size is estimated by counting the number of source instructions in the developed program. Obviously, while counting the number of source instructions, lines used for commenting the code and the header lines should be ignored.

Determining the LOC count at the end of a project is a very simple job. However, accurate estimation of the LOC count at the beginning of a project is very difficult. In order to estimate the LOC count at the beginning of a project, project managers usually divide the problem into modules, and each module into submodules and so on, until the sizes of the different leaf-level modules can be approximately predicted. To be able to do this, past experience in developing similar products is helpful. By using the estimation of the lowest level modules, project managers arrive at the total size estimation.

Function point (FP)

Function point metric overcomes many of the shortcomings of the LOC metric. Function point metric has been slowly gaining popularity. One of the important advantages of using the function point metric is that it can be used to easily estimate the size of a software product directly from the problem specification. This is in contrast to the LOC metric, where the size can be accurately determined only after the product has fully been developed. The conceptual idea behind the function point metric is that the size of a software product is

directly dependent on the number of different functions or features it supports. A software product supporting many features would certainly be of larger size than a product with less number of features. Each function when invoked reads some input data and transforms it to the corresponding output data. For example, the issue book feature (as shown in figure) of a Library Automation Software takes the name of the book as input and displays its location and the number of copies available. Thus, a computation of the number of input and the output data values to a system gives some indication of the number of functions supported by the system.

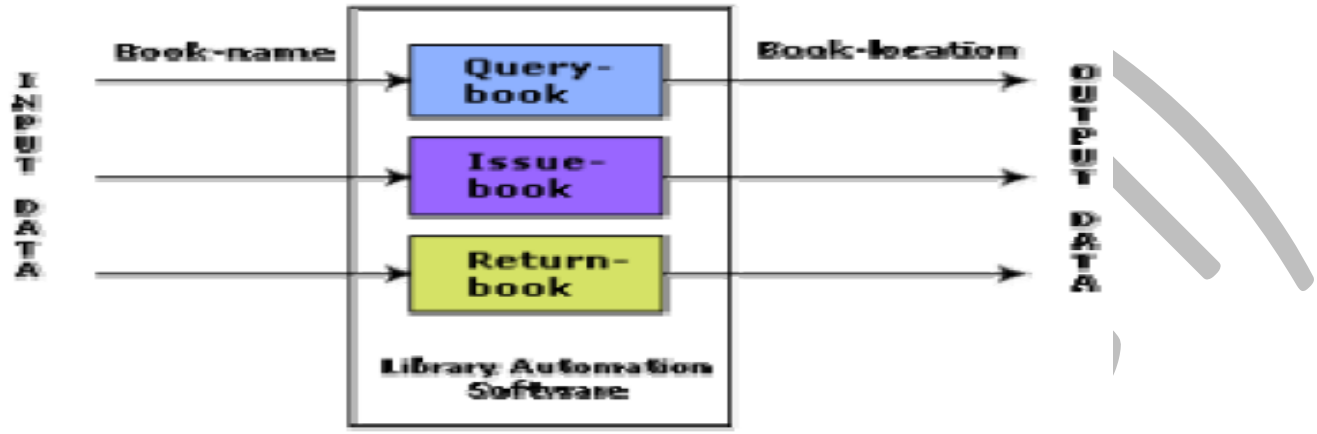


Fig. System function as a map of input data to output data

Besides using the number of input and output data values, function point metric computes the size of a software product (in units of functions points or FPs) using three other characteristics of the product as shown in the following expression. The size of a product in function points (FP) can be expressed as the weighted sum of these five problem characteristics. The weights associated with the five characteristics were proposed empirically and validated by the observations over many projects. Function point is computed in two steps. The first step is to compute the unadjusted function point (UFP).

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

Number of inputs: Each data item input by the user is counted. Data inputs should be distinguished from user inquiries. Inquiries are user commands such as print-account-balance.Inquiries are counted separately. It must be noted that individual data items input by the user are not considered in the calculation of the number of inputs, but a group of related inputs are considered as a single input. For example, while entering the data concerning an employee to an employee pay roll software; the data items name, age, sex, address, phone number, etc. are together considered as a single input. All these data items can be considered to be related, since they pertain to a single employee.

Number of outputs: The outputs considered refer to reports printed, screen outputs, error messages produced, etc. While outputting the number of outputs the individual data items within a report are not considered, but a set of related data items is counted as one input.

Number of inquiries: Number of inquiries is the number of distinct interactive queries which can be made by the users. These inquiries are the user commands which require specific action by the system.

Number of files: Each logical file is counted. A logical file means groups of logically related data. Thus, logical files can be data structures or physical files.

Number of interfaces: Here the interfaces considered are the interfaces used to exchange information with other systems. Examples of such interfaces are data files on tapes, disks, communication links with other systems etc.

Once the unadjusted function point (UFP) is computed, the technical complexity factor (TCF) is computed next. TCF refines the UFP measure by considering fourteen other factors such as high transaction rates, throughput, and response time requirements, etc. Each of these 14 factors is assigned from 0 (not present or no influence) to 6 (strong influence). The resulting numbers are summed, yielding the total degree of influence (DI). Now, TCF is computed as (0.65+0.01*DI). As DI can vary from 0 to 70, TCF can vary from 0.65 to 1.35. Finally, FP=UFP*TCF.

Shortcomings of function point (FP) metric

LOC as a measure of problem size has several shortcomings:

 LOC gives a numerical value of problem size that can vary widely with individual coding style – different programmers lay out their code in different ways. For example, one programmer might write several source instructions on a single line whereas another might split a single instruction across several lines. Of course, this problem can be easily overcome by counting the language tokens in the program rather than the lines of code.

 A good problem size measure should consider the overall complexity of the problem and the effort needed to solve it. That is, it should consider the local effort needed to specify, design, code, test, etc. and not just the coding effort. LOC, however, focuses on the coding activity alone; it merely computes the number of source lines in the final program.

 LOC measure correlates poorly with the quality and efficiency of the code. Larger code size does not necessarily imply better quality or higher efficiency. Some programmers produce lengthy and complicated code as they do not make effective use of the available instruction set.

 LOC metric penalizes use of higher-level programming languages, code reuse, etc. The paradox is that if a programmer consciously uses several library routines, then the LOC count will be lower.

 LOC metric measures the lexical complexity of a program and does not address the more important but subtle issues of logical or structural complexities. Between two programs with equal LOC count, a program having complex logic would require much more effort to develop than a program with very simple logic.

 It is very difficult to accurately estimate LOC in the final product from the problem specification. The LOC count can be accurately computed only after the code has been fully developed.

Feature Point Metric

A major shortcoming of the function point measure is that it does not take into account the algorithmic complexity of a software. That is, the function point metric implicitly assumes that the effort required to design and develop any two functionalities of the system is the same. But, we know that this is normally not true, the effort required to develop any two functionalities may vary widely. It only takes the number of functions that the system supports into consideration without distinguishing the difficulty level of developing the various functionalities. To overcome this problem, an extension of the function point metric called feature point metric is proposed.

Feature point metric incorporates an extra parameter algorithm complexity. This parameter ensures that the computed size using the feature point metric reflects the fact that the more is

the complexity of a function, the greater is the effort required to develop it and therefore its size should be larger compared to simpler functions.

Project Estimation Techniques --

Estimation of various project parameters is a basic project planning activity. The important project parameters that are estimated include: project size, effort required to develop the software, project duration, and cost. These estimates not only help in quoting the project cost to the customer, but are also useful in resource planning and scheduling. There are three broad categories of estimation techniques:

- Empirical estimation techniques
- Heuristic techniques
- Analytical estimation techniques

Empirical Estimation Techniques

Empirical estimation techniques are 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, different activities involved in estimation have been formalized over the years. Two popular empirical estimation techniques are: ***Expert judgment technique*** and ***Delphi cost estimation.***

Expert Judgment Technique

Expert judgment is one of the most widely used estimation techniques. In this approach, an expert makes an educated guess of the problem size after analyzing the problem thoroughly. Usually, the expert estimates the cost of the different components (i.e. modules or subsystems) of the system and then combines them to arrive at the overall estimate. However, this technique is subject to human errors and individual bias. Also, it is possible that the expert may overlook some factors inadvertently. Further, an expert making an estimate may not have experience and knowledge of all aspects of a project. For example, he may be conversant with the database and user interface parts but may not be very knowledgeable about the computer communication part. A more refined form of expert judgment is the estimation made by group of experts. Estimation by a group of experts minimizes factors 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. However, the estimate made by a group of experts may still exhibit bias on issues where the entire group of experts may be biased due to reasons such as political considerations. Also, the decision made by the group may be dominated by overly assertive members.

Delphi Cost Estimation

Delphi cost estimation approach tries to overcome some of the shortcomings of the expert judgment approach. Delphi estimation is carried out by a team comprising of 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 to the coordinator. In their estimates, the estimators mention any unusual characteristic of the product which has influenced his estimation. The coordinator prepares and distributes the summary of the responses of all the estimators, and includes any unusual rationale noted by any of the estimators. Based on this summary, the estimators re-estimate. This process is iterated for several rounds. However, no discussion among the estimators is allowed during the entire estimation process. The idea behind this 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.

HEURISTIC TECHNIQUES

Heuristic techniques assume that the relationships among the different project parameters can be modeled using suitable mathematical expressions. Once the basic (independent) parameters are known, the other (dependent) parameters can be easily determined by substituting the value of the basic parameters in the mathematical expression. Different heuristic estimation models can be divided into the following two classes: single variable model and the multi variable model.

Single variable estimation models provide a means to estimate the desired characteristics of a problem, using some previously estimated basic (independent) characteristic of the software product such as its size. A single variable estimation model takes the following form:

Estimated Parameter = $\mathbf{c1 * e^{d1}}$

In the above expression, e is the characteristic of the software which has already been estimated (independent variable). *Estimated Parameter* is the dependent parameter to be estimated. The dependent parameter to be estimated could be effort, project duration, staff size, etc. c1 and d1 are constants. The values of the constants c1 and d1 are usually determined using data collected from past projects (historical data). The basic COCOMO model is an example of single variable cost estimation model.

A multivariable cost estimation model takes the following form:

Estimated Resource = $\mathbf{c1*e1^{d1} + c2*e2^{d2} + ...}$

Where e1, e2, … are the basic (independent) characteristics of the software already estimated, and c1, c2, d1, d2, … are constants. Multivariable estimation models are expected to give more accurate estimates compared to the single variable models, since a project parameter is typically influenced by several independent parameters. The independent parameters influence the dependent parameter to different extents. This is modeled by the constants c1, c2, d1, d2, … . Values of these constants are usually determined from historical data. The intermediate COCOMO model can be considered to be an example of a multivariable estimation model.

Analytical Estimation Techniques

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 value, actual volume, effort, and development time.

For a given program, let:

- η1 be the number of unique operators used in the program,
- η2 be the number of unique operands used in the program,
- N1 be the total number of operators used in the program,
- N2 be the total number of operands used in the program.

Length and Vocabulary

The length of a program as defined by Halstead, quantifies total usage of all operators and operands in the program. Thus, length N = N1 +N2. Halstead's definition of the length of the program as the total number of operators and operands roughly agrees with the intuitive notation of the program length as the total number of tokens used in the program.

The program vocabulary is the number of unique operators and operands used in the program. Thus, *program vocabulary* $\eta = \eta1 + \eta2$.

Program Volume

$V = N\log2\eta$

Here the program volume V is the minimum number of bits needed to encode the program. In fact, to represent η different identifiers uniquely, at least log2η bits (where η is the program vocabulary) will be needed. In this scheme, Nlog2η bits will be needed to store a program of length N. Therefore, the volume V represents the size of the program by approximately compensating for the effect of the programming language used.

Length Estimation

Even though the length of a program can be found by calculating the total number of operators and operands in a program, Halstead suggests a way to determine the length of a program using the number of unique operators and operands used in the program. Using this method, the program parameters such as length, volume, cost, effort, etc. can be determined even before the start of any programming activity.

$$\mathbf{N = \log2\eta1^{\eta1} + \log2\eta2^{\eta2}}$$
$$\mathbf{= \eta1\log2\eta1 + \eta2\log2\eta2}$$

Example:

Let us consider the following C program:

```
main( )
{
int a, b, c, avg;
scanf("%d %d %d", &a, &b, &c);
avg = (a+b+c)/3;
printf("avg = %d", avg);
}
```

The unique operators are:

main,(),{},int,scanf,&,",",";",=,+,/, printf

The unique operands are:

a, b, c, &a, &b, &c, a+b+c, avg, 3,
"%d %d %d", "avg = %d"

Therefore,

η1 = 12, η2 = 11

Estimated Length = **(12*log12 + 11*log11)**
= (12*3.58 + 11*3.45)
= (43+38) = 81

Volume = **Length*log(23)**
= 81*4.52
= 366

COCOMO MODEL --

Organic, Semidetached and Embedded software projects

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. In order to classify a product into the identified categories, Boehm not only

considered the characteristics of the product but also those of the development team and development environment. Boehm's [1981] definition of organic, semidetached, and embedded systems are elaborated below.

Organic: A development project can be considered 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 considered of semidetached type, if the development 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 complex hardware, or if the stringent regulations on the operational procedures exist.

COCOMO

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm [1981]. According to Boehm, software cost estimation should be done through three stages: Basic COCOMO, Intermediate COCOMO, and Complete COCOMO.

Basic COCOMO Model

The basic COCOMO model gives an approximate estimate of the project parameters. The basic COCOMO estimation model is given by the following expressions:

Effort = a1 x (KLOC)^{a2} PM

Tdev = b1 x (Effort)^{b2} Months

Where

- KLOC is the estimated size of the software product expressed in Kilo Lines of Code,
- a1, a2, b1, b2 are constants for each category of software products,
- 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).

The effort estimation is expressed in units of person-months (PM). It is the area under the person-month plot (as shown in figure). It should be carefully noted that an effort of 100 PM does not imply that 100 persons should work for 1 month nor does it imply that 1 person should be employed for 100 months, but it denotes the area under the person-month curve (as shown in figure).

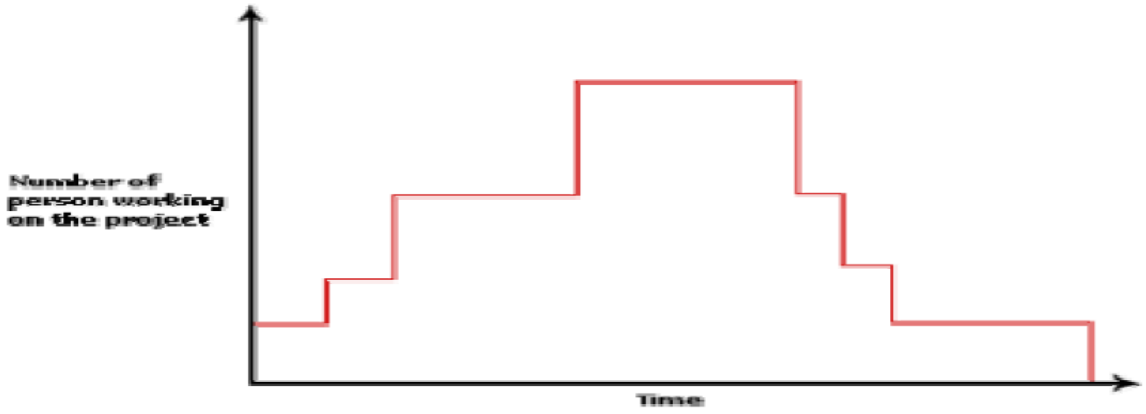


Fig. Person-month curve

According to Boehm, every line of source text should be calculated as one LOC irrespective of the actual number of instructions on that line. Thus, if a single instruction spans several lines (say n lines), it is considered to be nLOC. The values of a1, a2, b1, b2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm [1981] are summarized below. He derived the above expressions by examining historical data collected from a large number of actual projects.

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**

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Figure shows a plot of estimated effort versus product size. From figure , we can observe that the effort is somewhat super linear in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

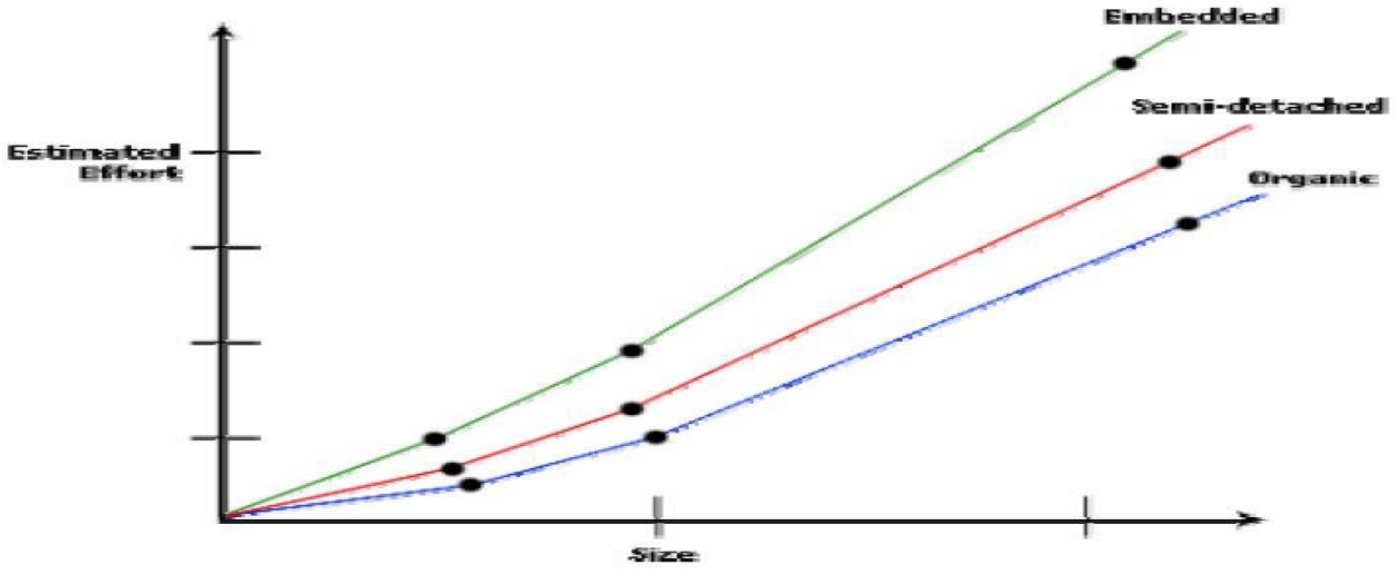


Fig. Effort versus product size

The development time versus the product size in KLOC is plotted in figure. From figure below, it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project. Further, from figure, it can be observed that the development time is roughly the same for all the three categories of products. For example, a

60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

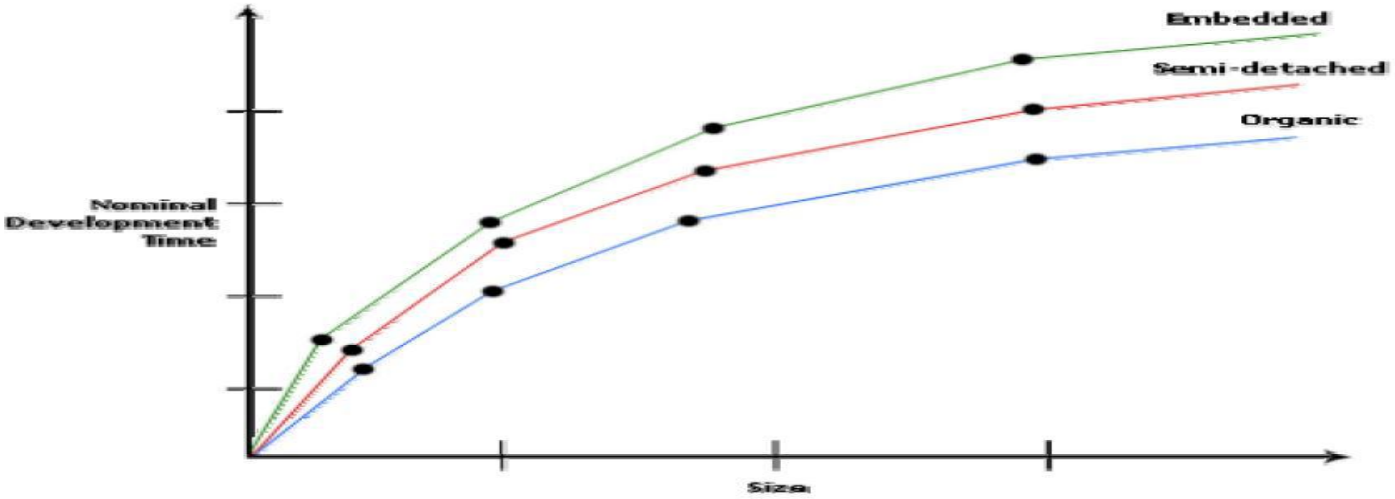


Fig. Development time versus size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called as nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example:

Assume that the size of an org organic type software product has been estimated to be 32,000 lines of source code. Assume that the average salary of software engineers be Rs. 15,000/- per month. Determine the effort required to develop the software product and the nominal development time.

From the basic COCOMO estimation formula for organic software:

Effort = **$2.4 \times (32)^{1.05} = 91$ PM**

Nominal development time = **$2.5 \times (91)^{0.38} = 14$ months**

Cost required to develop the product = **$14 \times 15{,}000$**

= **Rs. 210,000/-**

INTERMEDIATE COCOMO MODEL--

The intermediate COCOMO model recognizes this fact and refines the initial estimate obtained using the basic COCOMO expressions by using a set of 15 cost drivers (multipliers) based on various attributes of software development. For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1. If there are stringent reliability requirements on the software product, this initial estimate is scaled upward. Boehm requires the project manager to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, he suggests appropriate cost driver values which

should be multiplied with the initial estimate obtained using the basic COCOMO. In general, the cost drivers 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, 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 model--

A major shortcoming of both the basic and intermediate COCOMO models is that they consider a software product as a single homogeneous entity. However, most large systems are made up several smaller sub-systems. These sub-systems may have widely different characteristics. For example, some sub-systems may be considered as organic type, some semidetached, and some embedded. Not only that the inherent development complexity of the subsystems may be different, but also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development, and so on. 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 subsystems. The cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

The following development project can be considered as an example application of the complete COCOMO model. A distributed Management Information System (MIS) product for an organization having offices at several places across the country can have the following sub-components:

• Database part
• Graphical User Interface (GUI) part
• Communication part

Of these, the communication part can be considered as embedded software. The database part could be semi-detached software, and the GUI part organic software. The costs for these three components can be estimated separately, and summed up to give the overall cost of the system.

PROJECT SCHEDULING

Project-task scheduling is an important project planning activity. It involves deciding which tasks would be taken up when. In order to schedule the project activities, a software project manager needs to do the following:

1. Identify all the tasks needed to complete the project.
2. Break down large tasks into small activities.
3. Determine the dependency among different activities.
4. Establish the most likely estimates for the time durations necessary to complete the activities.
5. Allocate resources to activities.
6. Plan the starting and ending dates for various activities.

7. Determine the critical path. A critical path is the chain of activities that determines the duration of the project.

The first step in scheduling a software project involves identifying all the tasks 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 tasks of the project. Next, the large tasks are broken down into a logical set of small activities which would be assigned to different engineers. The work breakdown structure formalism helps the manager to breakdown the tasks systematically after the project manager has broken down the tasks and created the work breakdown structure, he has to find the dependency among the activities. Dependency among the different activities determines the order in which the different activities would be carried out. If an activity A requires the results of another activity B, then activity A must be scheduled after activity B. In general, the task dependencies define a partial ordering among tasks, i.e. each tasks may precede a subset of other tasks, but some tasks might not have any precedence ordering defined between them (called concurrent task). The dependency among the activities is represented in the form of an activity network.

Once the activity network representation has been worked out, resources are allocated to each activity. Resource allocation is typically done using a Gantt chart. After resource allocation is done, a PERT chart representation is developed. The PERT chart representation is suitable for program monitoring and control. For task scheduling, the project manager needs to decompose the project tasks into a set of activities. The time frame when each activity is to be performed is to be determined. The end of each activity is called milestone. The project manager tracks the progress of a project by monitoring the timely completion of the milestones. If he observes that the milestones start getting delayed, then he has to carefully control the activities, so that the overall deadline can still be met.

Work Breakdown Structure

Work Breakdown Structure (WBS) is used to decompose a given task set recursively into small activities. WBS provides a notation for representing the major tasks need to be carried out in order to solve a problem. The root of the tree is labeled by the problem name. Each node of the tree is broken down into smaller activities that are made the children of the node. Each activity is recursively decomposed into smaller sub-activities until at the leaf level, the activities requires approximately two weeks to develop. Fig. 36.1 represents the WBS of a MIS (Management Information System) software.

While breaking down a task into smaller tasks, the manager has to make some hard decisions. If a task is broken down into large number of very small activities, these can be carried out independently. Thus, it becomes possible to develop the product faster (with the help of additional manpower). Therefore, to be able to complete a project in the least amount of time, the manager needs to break large tasks into smaller ones, expecting to find more parallelism. However, it is not useful to subdivide tasks into units which take less than a week or two to execute. Very fine subdivision means that a disproportionate amount of time must be spent on preparing and revising various charts.

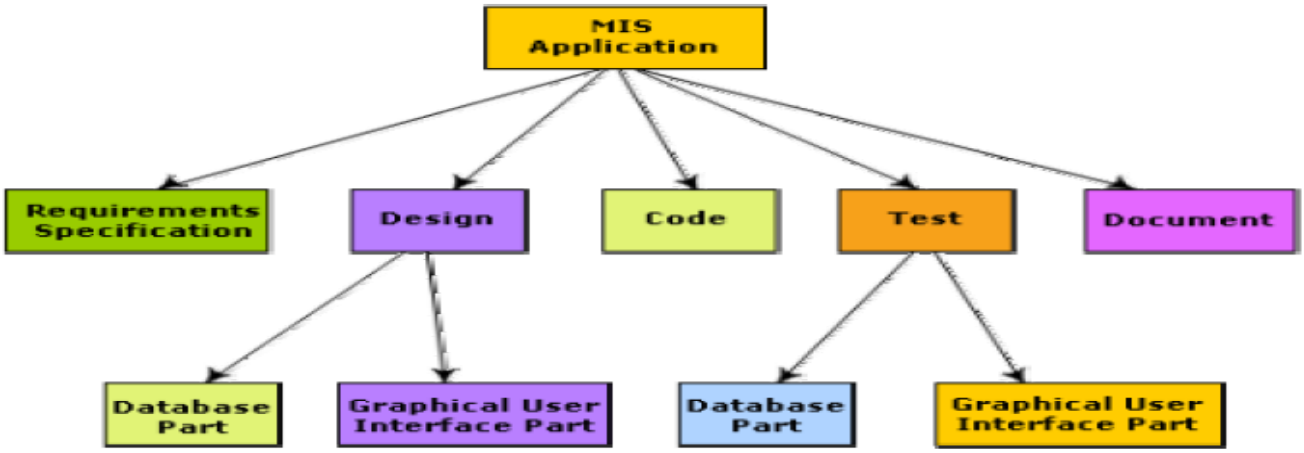


Fig. Work breakdown structure of an MIS problem

Activity networks and critical path method WBS representation of a project is transformed into an activity network by representing activities identified in WBS along with their interdependencies. An activity network shows the different activities making up a project, their estimated durations, and interdependencies (as shown in figure below). Each activity is represented by a rectangular node and the duration of the activity is shown alongside each task.

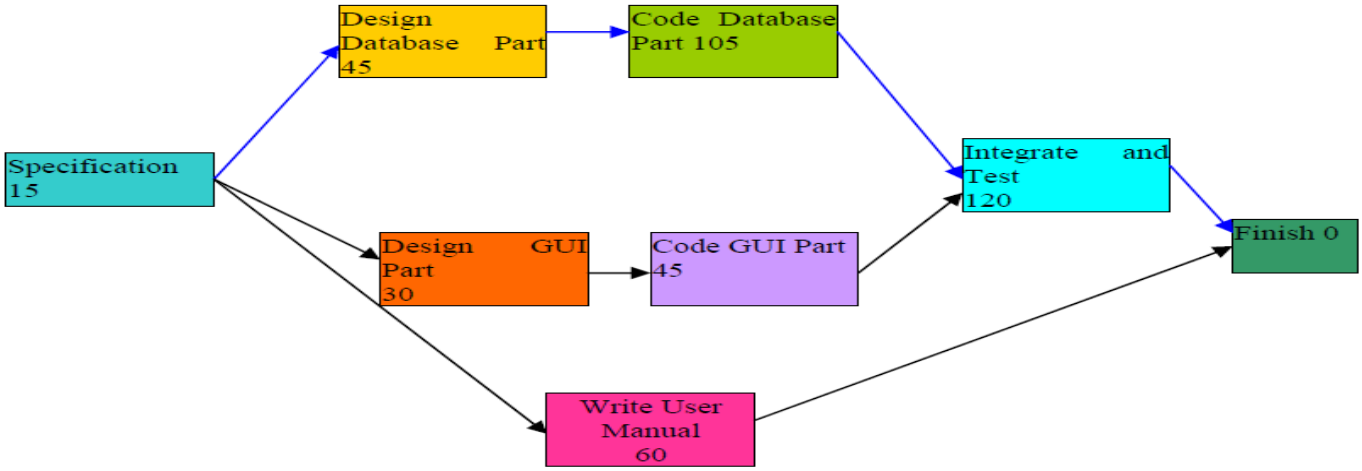


Fig. Activity network representation of the MIS problem

Critical Path Method (CPM)

From the activity network representation following analysis can be made. The minimum time (MT) to complete the project is the maximum of all paths from start to finish. The earliest start (ES) time of a task is the maximum of all paths from the start to the task. The latest start (LS) time is the difference between MT and the maximum of all paths from this task to the finish. The earliest finish time (EF) of a task is the sum of the earliest start time of the task and the duration of the task. The latest finish (LF) time of a task can be obtained by subtracting maximum of all paths from this task to finish from MT. The slack time (ST) is LS – ES and equivalently can be written as LF – EF. 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. A critical task is one with a zero slack time. A path from the start node to the finish node containing only critical tasks is called a critical path. These parameters for different tasks for the MIS problem are shown in the following table.

Task	ES	EF	LS	LF	ST
Specification	0	15	0	15	0
Design database	15	60	15	60	0
Design GUI part	15	45	90	120	75
Code database	60	165	60	165	0
Code GUI part	45	90	120	165	75
Integrate and test	165	285	165	285	0
Write user manual	15	75	225	285	210

The critical paths are all the paths whose duration equals MT. The critical path in figure is shown with a dark arrow.

Gantt Chart

Gantt charts are mainly used to allocate resources to activities. The resources allocated to activities include staff, hardware, and software. Gantt charts (named after its developer Henry Gantt) are useful for resource planning. A Gantt chart is a special type of bar chart where each bar represents an activity. The bars are drawn along a time line. The length of each bar is proportional to the duration of time planned for the corresponding activity.

Gantt charts are 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 white part and a shaded part. The shaded part of the bar shows the length of time each task is estimated to take. The white part shows the slack time, that is, the latest time by which a task must be finished. A Gantt chart representation for the MIS problem is shown in the figure below.

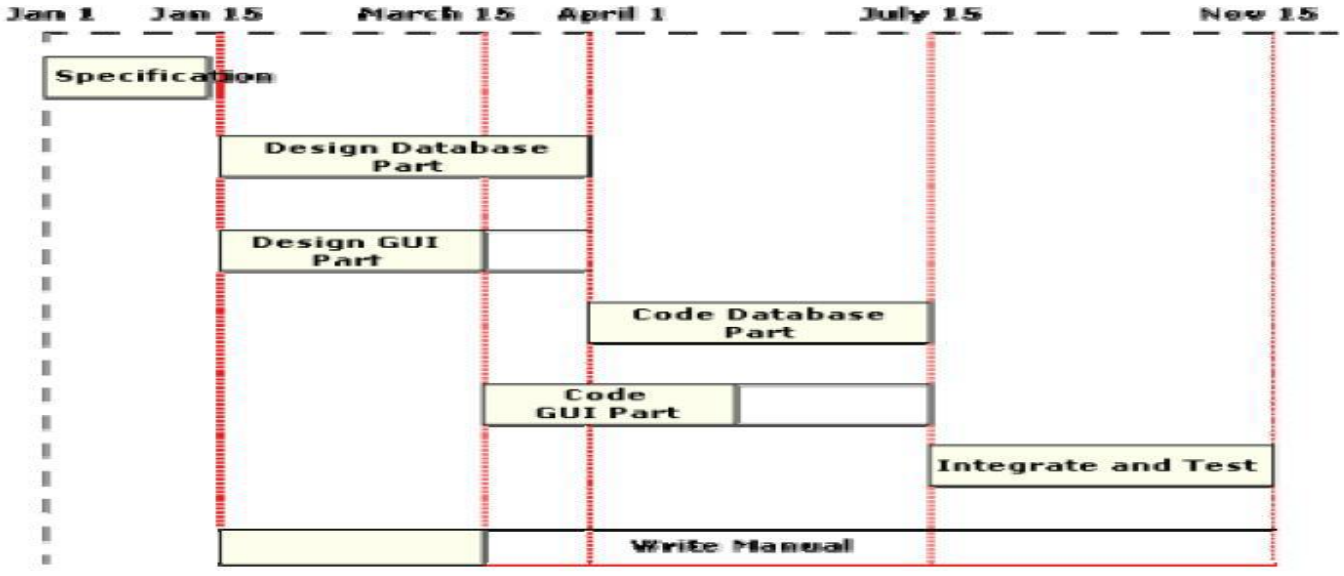


Fig. Gantt chart representation of the MIS problem

PERT Chart

PERT (Project Evaluation and Review Technique) charts consist of a network of boxes and arrows. The boxes represent activities and the arrows represent task dependencies. PERT chart represents the statistical variations in the project estimates assuming a normal distribution. Thus, in a PERT chart instead of making a single estimate for each task, pessimistic, likely, and optimistic estimates are made. The boxes of PERT charts are usually

annotated with the pessimistic, likely, and optimistic estimates for every task. Since all possible completion times between the minimum and maximum duration for every task has to be considered, there are not one but many critical paths, depending on the permutations of the estimates for each task. This makes critical path analysis in PERT charts very complex. A critical path in a PERT chart is shown by using thicker arrows. The PERT chart representation of the MIS problem is shown in figure below. PERT charts are a more sophisticated form of activity chart. In activity diagrams only the estimated task durations are represented. Since, the actual durations might vary from the estimated durations, the utility of the activity diagrams are limited. 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 engineers.

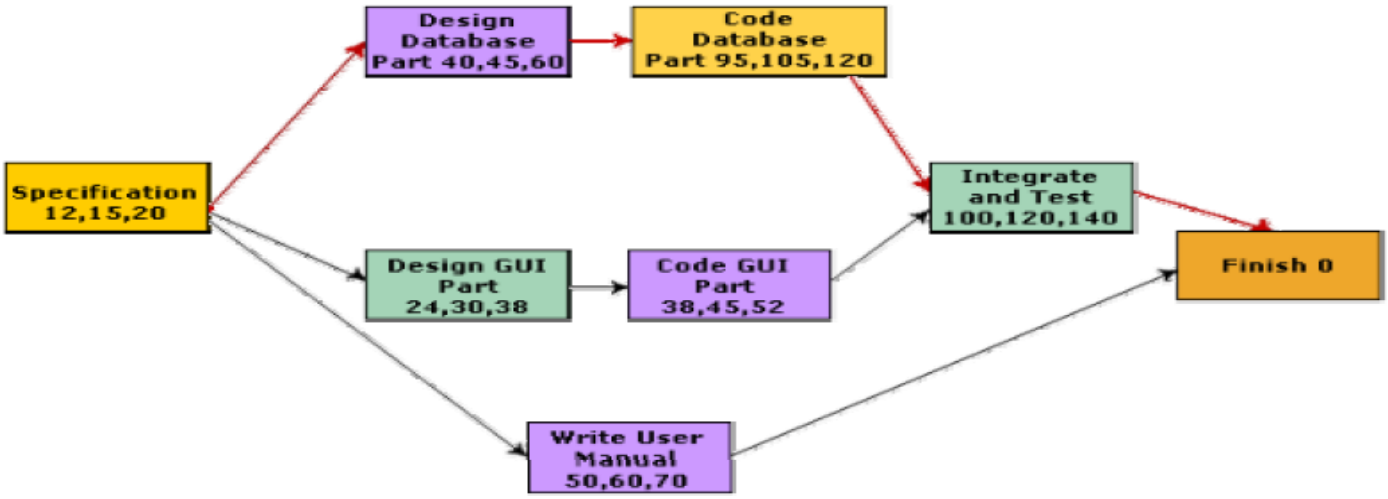


Fig. PERT chart representation of the MIS problem

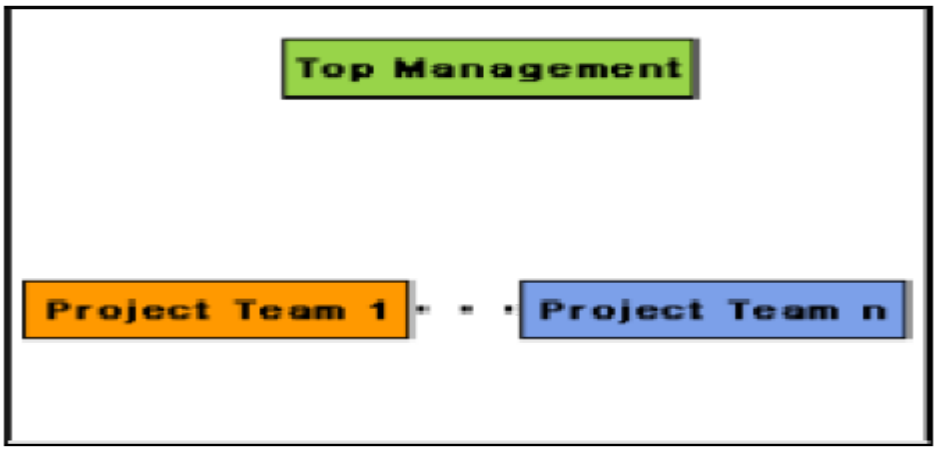
ORGANIZATION STRUCTURE

Functional format vs. project format

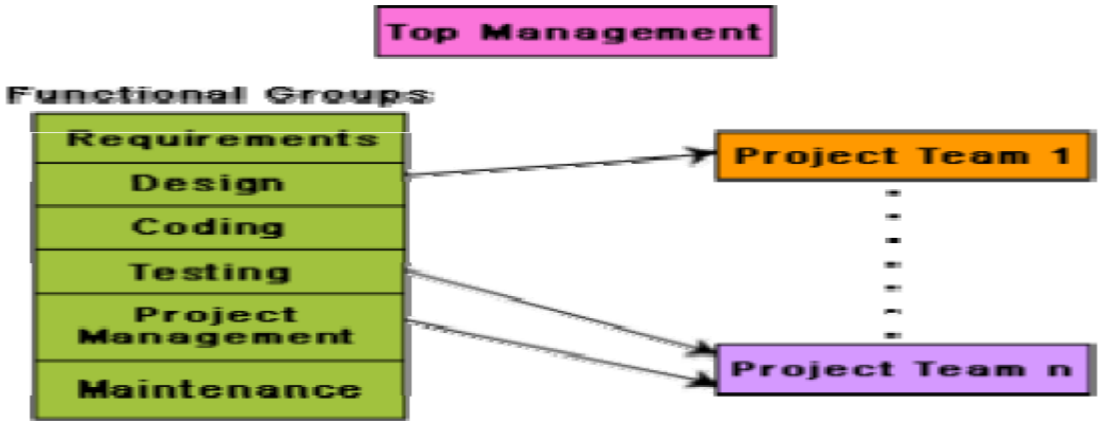
There are essentially two broad ways in which a software development organization can be structured: functional format and project format. In the project format, the project development staffs are divided based on the project for which they work (as shown in figure). In the functional format, the development staffs are divided based on the functional group to which they belong. The different projects borrow engineers from the required functional groups for specific phases to be undertaken in the project and return them to the functional group upon the completion of the phase.

a) Project Organization

b) Functional Organization



a) Project Organization



b) Functional Organization

Fig. Schematic representation of the functional and project organization

In the functional format, different teams of programmers perform different phases of a project. For example, one team might do the requirements specification, another do the design, and so on. The partially completed product passes from one team to another as the project evolves. Therefore, the functional format requires considerable communication among the different teams because the work of one team must be clearly understood by the subsequent teams working on the project. This requires good quality documentation to be produced after every activity.

In the project format, a set of engineers is assigned to the project at the start of the project and they remain with the project till the completion of the project. Thus, the same team carries out all the life cycle activities. Obviously, the functional format requires more communication among teams than the project format, because one team must understand the work done by the previous teams.

Advantages of functional organization over project organization

The main advantages of a functional organization are:

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

The functional organization allows the engineers to become specialists in particular roles, e.g. requirements analysis, design, coding, testing, maintenance, etc. They perform these roles again and again for different projects and develop deep insights to their work. It also results in more attention being paid to proper documentation at the end of a phase because of the greater need for clear communication as between teams doing different phases. The functional organization 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 organization. A project organization structure forces the manager to take in almost a constant number of engineers for the entire duration of his project. This results in engineers idling in the initial phase of the software development and are under tremendous pressure in the later phase of the development. A further advantage of the functional organization is that it is more effective in handling the problem of manpower turnover. This is because engineers can be brought in from the functional pool when needed. Also, this organization mandates production of good quality documents, so new engineers can quickly get used to the work already done.

Unsuitability of functional format in small organizations

In spite of several advantages of the functional organization, it is not very popular in the software industry. The project format provides job rotation to the team members. That is, each team member takes on the role of the designer, coder, tester, etc during the course of the project. On the other hand, considering the present skill shortage, it would be very difficult for the functional organizations to fill in slots for some roles such as maintenance, testing, and coding groups. Also, another problem with the functional organization is that if an organization handles projects requiring knowledge of specialized domain areas, then these domain experts cannot be brought in and out of the project for the different phases, unless the company handles a large number of such projects. Also, for obvious reasons the functional format is not suitable for small organizations handling just one or two projects.

Team Structures

Team structure addresses the issue of organization of the individual project teams. There are some possible ways in which the individual project teams can be organized. There are mainly three formal team structures: chief programmer, democratic, and the mixed team organizations although several other variations to these structures are possible. Problems of different complexities and sizes often require different team structures for chief solution.

Chief Programmer Team

In this team organization, a senior engineer provides the technical leadership and is designated as the chief programmer. The chief programmer partitions the task into small activities and assigns them to the team members. He also verifies and integrates the products developed by different team members. The structure of the chief programmer team is shown in figure. The chief programmer provides an authority, and this structure is arguably more efficient than the democratic team for well-understood problems. However, the chief programmer team leads to lower team morale, since team-members work under the constant supervision of the chief programmer. This also inhibits their original thinking. The chief programmer team is subject to single point failure since too much responsibility and authority is assigned to the chief programmer.

The chief programmer team is probably the most efficient way of completing simple and small projects since the chief programmer can work out a satisfactory design and ask the programmers to code different modules of his design solution.

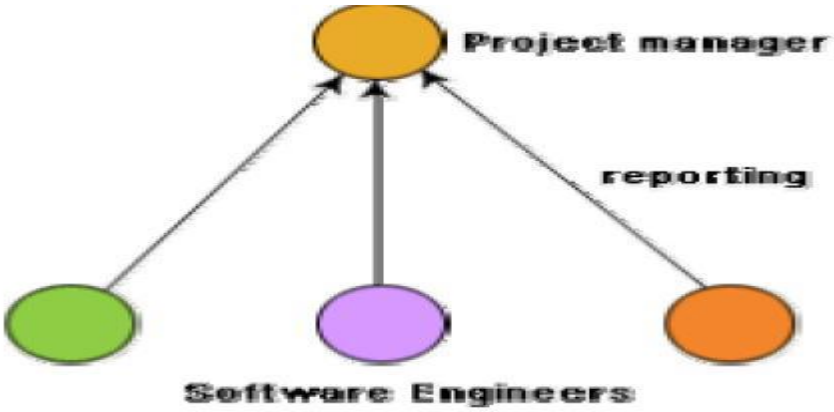


Fig. Chief programmer team structure

Democratic Team

The democratic team structure, as the name implies, does not enforce any formal team hierarchy (as shown in figure). Typically, a manager provides the administrative leadership. At different times, different members of the group provide technical leadership.

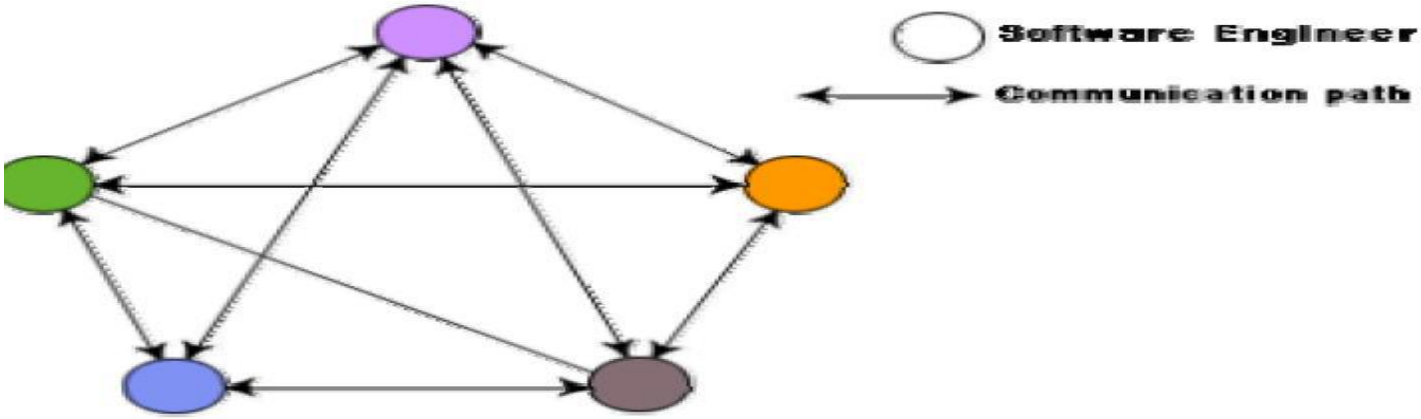


Fig. Democratic team structure

The democratic organization leads to higher morale and job satisfaction. Consequently, it suffers from less man-power turnover. Also, democratic team structure is appropriate for less understood problems, since a group of engineers can invent better solutions than a single individual as in a chief programmer team. A democratic team structure is suitable for projects requiring less than five or six engineers and for research-oriented projects. For large sized projects, a pure democratic organization tends to become chaotic. The democratic team organization encourages egoless programming as programmers can share and review one another's work.

Mixed Control Team Organization

The mixed team organization, as the name implies, draws upon the ideas from both the democratic organization and the chief-programmer organization. The mixed control team organization is shown pictorially in figure. This team organization incorporates both hierarchical reporting and democratic set up. In figure, the democratic connections are shown as dashed lines and the reporting structure is shown using solid arrows. The mixed control team organization is suitable for large team sizes. The democratic arrangement at the senior

engineers level is used to decompose the problem into small parts. Each democratic setup at the programmer level attempts solution to a single part. Thus, this team organization is eminently suited to handle large and complex programs. This team structure is extremely popular and is being used in many software development companies.

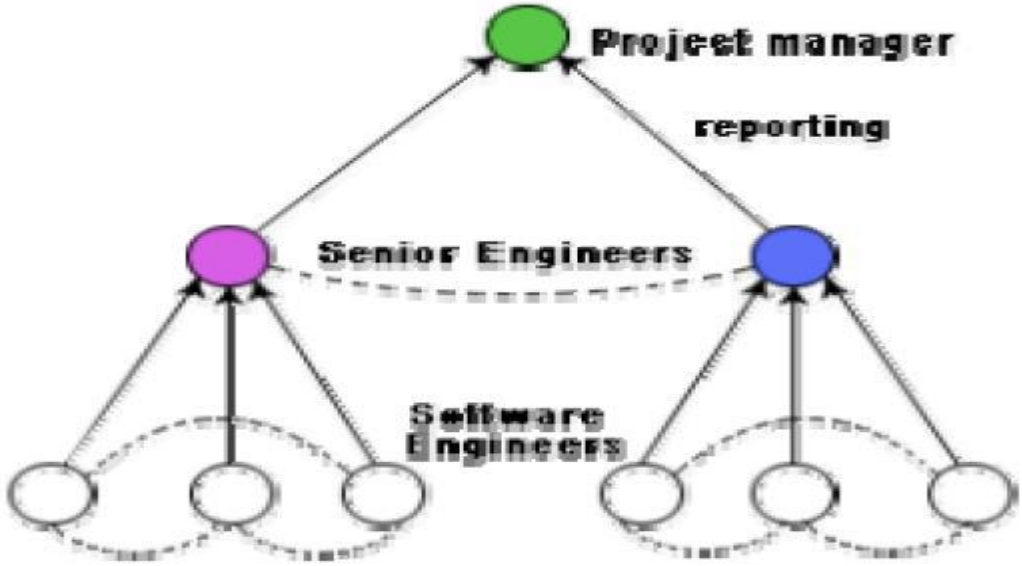


Fig. Mixed team structure

Egoless Programming Technique

Ordinarily, the human psychology makes an individual take pride in everything he creates using original thinking. Software development requires original thinking too, although of a different type. The human psychology makes one emotionally involved with his creation and hinders him from objective examination of his creations. Just like temperamental artists, programmers find it extremely difficult to locate bugs in their own programs or flaws in their own design. Therefore, the best way to find problems in a design or code is to have someone review it. Often, having to explain one's program to someone else gives a person enough objectivity to find out what might have gone wrong. This observation is the basic idea behind code walk throughs. An application of this, is to encourage a democratic team to think that the design, code, and other deliverables to belong to the entire group. This is called egoless programming technique.

STAFFING

Characteristics of a good software engineer

The attributes that good software engineers should possess are as follows:

- 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.

Studies show that these attributes vary as much as 1:30 for poor and bright candidates. An experiment conducted by Sackman [1968] shows that the ratio of coding hour for the worst to the best programmers is 25:1, and the ratio of debugging hours is 28:1. Also, the ability of a software engineer to arrive at the design of the software from a problem description varies greatly with respect to the parameters of quality and time.

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. A programmer having a thorough knowledge of database application (e.g. MIS) may turn out to be a poor data communication engineer. Lack of familiarity with the application areas can result in low productivity and poor quality of the product.

Since software development is a group activity, it is vital for a software engineer to possess three main kinds of communication skills: Oral, Written, and Interpersonal. A software engineer not only needs to effectively communicate with his teammates (e.g. reviews, walk throughs, and other team communications) but may also have to communicate with the customer to gather product requirements. Poor interpersonal skills hamper these vital activities and often show up as poor quality of the product and low productivity. Software engineers are also required at times to make presentations to the managers and to the customers. This requires a different kind of communication skill (oral communication skill). A software engineer is also expected to document his work (design, code, test, etc.) as well as write the users' manual, training manual, installation manual, maintenance manual, etc. This requires good written communication skill. Motivation level of software engineers 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 engineers may turn out to be poor performers when they have lack motivation. An average engineer who can work with a single mind track can outperform other engineers, higher incentives and better working conditions have only limited effect on their motivation levels. Motivation is to a great extent determined by personal traits, family and social backgrounds, etc.

Software Configuration Management --

The results (also called as the deliverables) of a large software development effort typically consist of a large number of objects, e.g. source code, design document, SRS document, test document, user's manual, etc. These objects are usually referred to and modified by a number of software engineers through out the life cycle of the software. The state of all these objects at any point of time is called the configuration of the software product. The state of each deliverable object changes as development progresses and also as bugs are detected and fixed.

Software Configuration Management Activities

The configuration management tool enables the engineers to change the various components in a controlled manner.

Configuration management is carried out through two principal activities:

• Configuration identification involves deciding which parts of the system should be kept track of.

• Configuration control ensures that changes to a system happen smoothly.

Configuration Identification

The project manager normally classifies the objects associated with a software development effort into three main categories: controlled, pre controlled, and uncontrolled. Controlled objects are those which are already put under configuration control. One must follow some formal procedures to change them. Pre controlled objects are not yet under configuration control, but will eventually be under configuration control. Uncontrolled objects are not and will not be subjected to configuration control. Controllable objects include both controlled and pre controlled objects. Typical controllable objects include:

• Requirements specification document

• Design documents

Tools used to build the system, such as compilers, linkers, lexical analyzers, parsers, etc.

• Source code for each module

• Test cases

• Problem reports

The configuration management plan is written during the project planning phase and it lists all controlled objects. The managers who develop the plan must strike a balance between controlling too much, and controlling too little. If too much is controlled, overheads due to configuration management increase to unreasonably high levels. On the other hand, controlling too little might lead to confusion when something changes.

Configuration Control

Configuration control is the process of managing changes to controlled objects. Configuration control is the part of a configuration management system that most directly affects the day-to-day operations of developers. The configuration control system prevents unauthorized changes to any controlled objects. 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 as shown in figure. Configuration management tools allow only one person to reserve a module at a time. Once an object is reserved, it does not allow anyone else to reserve this module until the reserved module is restored as shown in figure. Thus, by preventing more than one engineer to simultaneously reserve a module, the problems associated with concurrent access are solved.

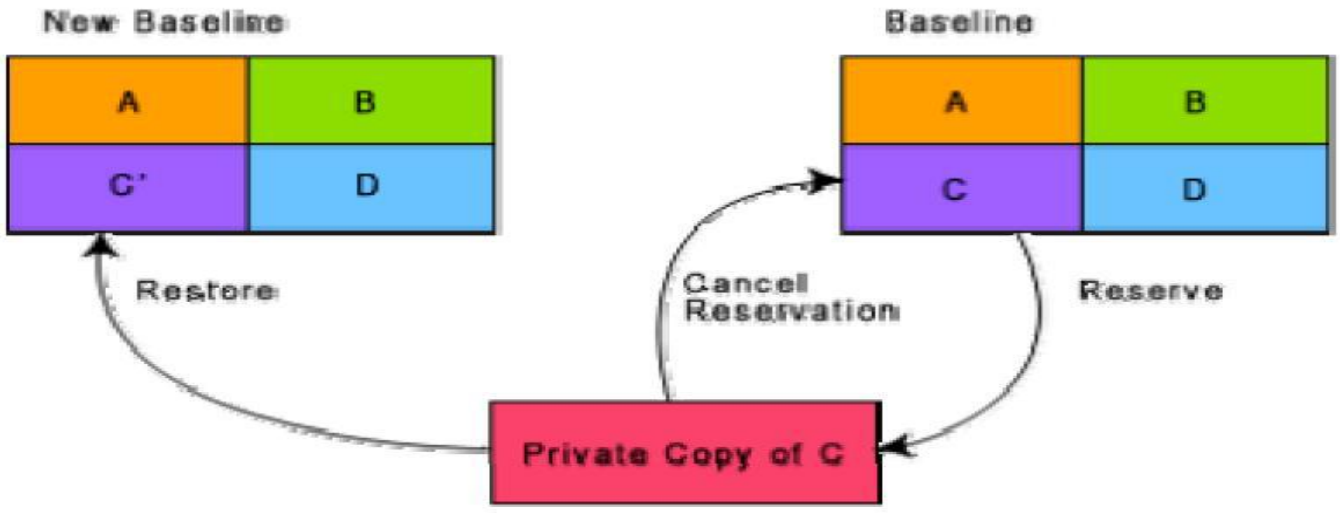


Fig. Reserve and restore operation in configuration control

It can be shown how the changes to any object that is under configuration control can be achieved. The engineer needing to change a module first obtains a private copy of the module through a reserve operation. Then, he carries out all necessary changes on this private copy. However, restoring the changed module to the system configuration requires the permission of a change control board (CCB). The CCB is usually constituted from among the development team members. For every change that needs to be carried out, the CCB reviews the changes made to the controlled object and certifies several things about the change:

1. Change is well-motivated.
2. Developer has considered and documented the effects of the change.
3. Changes interact well with the changes made by other developers.
4. Appropriate people (CCB) have validated the change, e.g. someone has tested the changed code, and has verified that the change is consistent with the requirement.

Risk Management--

A software project can be concerned with a large variety of risks. In order to be adept to systematically identify the significant risks which might affect a software project, it is essential to classify risks into different classes. The project manager can then check which risks from each class are relevant to the project.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

1. Project risks: Project risks concern differ forms of budgetary, schedule, personnel, resource, and customer-related problems. A vital project risk is schedule slippage. Since the software is intangible, it is very tough to monitor and control a software project. It is very tough to control something which cannot be identified. For any manufacturing program, such as the manufacturing of cars, the plan executive can recognize the product taking shape.

2. Technical risks: Technical risks concern potential method, implementation, interfacing, testing, and maintenance issue. It also consists of an ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks appear due to the development team's insufficient knowledge about the project.

3. Business risks: This type of risks contain risks of building an excellent product that no one need, losing budgetary or personnel commitments, etc.

Other risk categories

1. **1. Known risks:** Those risks that can be uncovered after careful assessment of the project program, the business and technical environment in which the plan is being developed, and more reliable data sources (e.g., unrealistic delivery date)
2. **2. Predictable risks:** Those risks that are hypothesized from previous project experience (e.g., past turnover)
3. **3. Unpredictable risks:** Those risks that can and do occur, but are extremely tough to identify in advance.

Principle of Risk Management

1. **Global Perspective:** In this, we review the bigger system description, design, and implementation. We look at the chance and the impact the risk is going to have.
2. **Take a forward-looking view:** Consider the threat which may appear in the future and create future plans for directing the next events.
3. **Open Communication:** This is to allow the free flow of communications between the client and the team members so that they have certainty about the risks.
4. **Integrated management:** In this method risk management is made an integral part of project management.
5. **Continuous process:** In this phase, the risks are tracked continuously throughout the risk management paradigm.

Risk Management Activities

Risk management consists of three main activities, as shown in fig:

Risk Assessment

The objective of risk assessment is to division the risks in the condition of their loss, causing potential. For risk assessment, first, every risk should be rated in two methods:

- The possibility of a risk coming true (denoted as r).
- The consequence of the issues relates to that risk (denoted as s).

Based on these two methods, the priority of each risk can be estimated:

$$p = r * s$$

Where p is the priority with which the risk must be controlled, r is the probability of the risk becoming true, and s is the severity of loss caused due to the risk becoming true. If all identified risks are set up, then the most likely and damaging risks can be controlled first, and more comprehensive risk abatement methods can be designed for these risks.

1. Risk Identification: The project organizer needs to anticipate the risk in the project as early as possible so that the impact of risk can be reduced by making effective risk management planning.

A project can be of use by a large variety of risk. To identify the significant risk, this might affect a project. It is necessary to categories into the different risk of classes.
There are different types of risks which can affect a software project:

1. **Technology risks:** Risks that assume from the software or hardware technologies that are used to develop the system.
2. **People risks:** Risks that are connected with the person in the development team.
3. **Organizational risks:** Risks that assume from the organizational environment where the software is being developed.
4. **Tools risks:** Risks that assume from the software tools and other support software used to create the system.
5. **Requirement risks:** Risks that assume from the changes to the customer requirement and the process of managing the requirements change.
6. **Estimation risks:** Risks that assume from the management estimates of the resources required to build the system

2. Risk Analysis: During the risk analysis process, you have to consider every identified risk and make a perception of the probability and seriousness of that risk.
There is no simple way to do this. You have to rely on your perception and experience of previous projects and the problems that arise in them.
It is not possible to make an exact, the numerical estimate of the probability and seriousness of each risk. Instead, you should authorize the risk to one of several bands:

1. The probability of the risk might be determined as very low (0-10%), low (10-25%), moderate (25-50%), high (50-75%) or very high (+75%).
2. The effect of the risk might be determined as catastrophic (threaten the survival of the plan), serious (would cause significant delays), tolerable (delays are within allowed contingency), or insignificant.

Risk Control
It is the process of managing risks to achieve desired outcomes. After all, the identified risks of a plan are determined; the project must be made to include the most harmful and the most likely risks. Different risks need different containment methods. In fact, most risks need ingenuity on the part of the project manager in tackling the risk.
There are three main methods to plan for risk management:

1. **Avoid the risk:** This may take several ways such as discussing with the client to change the requirements to decrease the scope of the work, giving incentives to the engineers to avoid the risk of human resources turnover, etc.
2. **Transfer the risk:** This method involves getting the risky element developed by a third party, buying insurance cover, etc.
3. **Risk reduction:** This means planning method to include the loss due to risk. For instance, if there is a risk that some key personnel might leave, new recruitment can be planned.

Risk Leverage: To choose between the various methods of handling risk, the project plan must consider the amount of controlling the risk and the corresponding reduction of risk. For this, the risk leverage of the various risks can be estimated.

Risk leverage is the variation in risk exposure divided by the amount of reducing the risk.
Risk leverage = (risk exposure before reduction - risk exposure after reduction) / (cost of reduction)

1. Risk planning: The risk planning method considers each of the key risks that have been identified and develop ways to maintain these risks.

For each of the risks, you have to think of the behavior that you may take to minimize the disruption to the plan if the issue identified in the risk occurs.

You also should think about data that you might need to collect while monitoring the plan so that issues can be anticipated.

Again, there is no easy process that can be followed for contingency planning. It rely on the judgment and experience of the project manager.

2. Risk Monitoring: Risk monitoring is the method king that your assumption about the product, process, and business risks has not changed.